Е.М.Воробьев

ВВЕДЕНИЕ В СИСТЕМУ "МАТЕМАТИКА"

Рассмотрены вопросы использования пакета "Математика" в качестве символьного, графического и численного калькулятора, а также языка программирования высокого уровня, позволяющего программировать в функциональном стиле. Использование пакета освобождает пользователей от многих рутинных математических операций.

Для преподавателей и студентов вузов, будет полезна также учащимся лицеев и гимназий.

Содержание

ПРЕДИСЛОВИЕ	3
Часть I. "МАТЕМАТИКА" КАК СИМВОЛЬНЫЙ, ГРАФИЧЕСКИЙ И ЧИСЛЕННЫЙ КАЛЬКУЛЯТОР	7
Глава 1. АЗБУКА "МАТЕМАТИКИ"	**8**
1.1. Первый сеанс	8
1.2. Основы синтаксиса "Математики"	14
1.3. Обзор "Математики"	18
Упражнения	33
Глава 2. СИМВОЛЬНЫЕ ВЫЧИСЛЕНИЯ	**34**
2.1. Преобразования многочленов	34
2.2. Подстановки	38
2.3. Преобразования рациональных выражений	40
2.4. Предикаты и булевы операции	42
2.5. Алгебраические и трансцендентные уравнения	45
2.6. Математический анализ	51
2.7. Специализированные программы	56
2.8. Обыкновенные дифференциальные уравнения	58
2.9. Числа и операции над числами	64
Упражнения	68
Глава 3. ВСТРОЕННАЯ ГРАФИКА	**70**
3.1. Графические функции и их опции	70
3.2. Двумерная графика	76
3.3. Трехмерная графика	83
3.4. Изменение стиля и комбинирование построенных рисунков	87
3.5. Мультипликация	88
3.6. Графические функции специализированных пакетов	89
3.7. Графические примитивы	92
Упражнения	98
Глава 4. РАБОТА СО СПИСКАМИ	**100**
4.1. Порождение списков	100
4.2. Преобразования списков	103
4.3. Работа с векторами и матрицами	110
4.4. Выражения "Математики"	113
4.5. Вычисление функций от списков и их элементов	117

Упражнения 123
Часть II. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМПЬЮТЕРНОЙ АЛГЕБРЫ "МАТЕМАТИКА" 125
Глава 5. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ 126
5.1. Функции, определяемые пользователем 126
5.2. Чистые и анонимные функции 129
5.3. Суперпозиция функций 132
5.4. Подмножества конечного множества 136
Упражнения 138
Глава 6. ПРОГРАММИРОВАНИЕ, ОСНОВАННОЕ НА ПРАВИЛАХ ПРЕОБРАЗОВАНИЙ 140
6.1. Глобальные и локальные правила преобразований 141
6.2. Шаблоны 148
6.3. Шаблоны в глобальных правилах преобразований 152
6.4. Шаблоны в локальных правилах преобразований 158
Упражнения 162
Глава 7. ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ 164
7.1. Составные выражения. Оператор Do 164
7.2. Условные операторы 167
7.3. Условные циклы 171
7.4. Функция Module 173
Упражнения 175
Глава 8. ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЙ 177
8.1. Значения, ассоциированные с символами 178
8.2. Атрибуты 179
8.3. Стандартный процесс вычислений 185
8.4. Выражения, вычисляемые нестандартно 187
8.5. Вычисление правил преобразований 191
Упражнения 194
Глава 9. РАЗРАБОТКА ПРОГРАММ 195
9.1. Контексты 195
9.2. Контексты и программы 201
9.3. Подгрузка программ 204
Упражнения 207
Глава 10. ВВОД И ВЫВОД ДАННЫХ 209
10.1. Ввод и запись данных в файлы 209
10.2. Обмен данными с другими программами 216
10.3. Форматирование выходных ячеек 218
Упражнения 224
Ответы и решения к упражнениям 226
Краткий справочник по встроенным функциям "Математики" 232
Предметный указатель 252
Литература 258

ПРЕДИСЛОВИЕ

Длительная эволюция применения компьютеров для численных расчетов привела к развитию методов компьютерного моделирования и вычислительного эксперимента. Активное использование компьютеров для проведения символьных и графических вычислений, освобождающее исследователя от проведения рутинных, но трудоемких и чреватых ошибками преобразований, существенно сократило время реализации научных и технических проектов. Взрывное развитие компьютерных телекоммуникаций позволяет отдельным исследователям получать доступ к таким информационным и вычислительным ресурсам, которыми ранее располагали только крупные научные организации.

В контекст рассмотренных изменений органично вписывается программный продукт „Система компьютерной алгебры «Математика»“ американской фирмы Wolfram Research, Inc. (первая версия — 1988 г., третья — 1996 г.). Слова „компьютерная алгебра“ являются скорее данью традиции, так как с помощью „Математики“ можно осуществлять широкий спектр символьных преобразований, включающий, наряду с другими, операции математического анализа: дифференцирование, интегрирование и интегральные преобразования, разложение в ряды, решение дифференциальных уравнений и т.п. Одна из сильных сторон рассматриваемого программного про-

дукта — развитая двух- и трехмерная графика, используемая для визуализации математических объектов.

По своей сущности „Математика“ представляет собой язык программирования высокого уровня, позволяющий реализовать традиционный процедурный и функциональный стили программирования, а также стиль правил преобразований. Поскольку рассматриваемый программный продукт обеспечивает также применение разнообразных численных методов, то в совокупности символьные, графические и численные вычисления, выполняемые в одном сеансе использования „Математики“, превращают ее в удобный и мощный инструмент математических исследований. Фактическое разделение в последних версиях вычислительного ядра и интерфейса, взаимодействующих между собой на основе протокола MathLink, позволяет использовать удаленные ядра в локальных и глобальных вычислительных сетях. Большое внимание уделяется в последнее время разработке программных средств сопряжения „Математики“ с Интернет.

В настоящее время „Математика“ является одной из компонент компьютерных технологий проведения научных исследований и обучения студентов в высшей школе США, Западной Европы и Японии. Использование „Математики“ для быстрых и безошибочных вычислений и для визуализации позволяет исследователям сосредоточить свое внимание на концептуальных вопросах научных проектов. „Математика“ также обеспечивает преподавание, основанное на огромном разнообразии реальных примеров математического, естественнонаучного, технического и гуманитарного характера. Сложилось международное сообщество пользователей этого программного продукта. Список научных монографий, посвященных приложениям

„Математики“, содержит в настоящее время более двух сотен наименований, издаются два журнала, проводятся ежегодные научные конференции. Фирма-разработчик поддерживает электронный архив (www.wolfram.com/mathSource/), содержащий в свободном доступе огромное количество написанных пользователями и сотрудниками фирмы научных, методических и учебных программных продуктов.

Мы надеемся, что предлагаемая вниманию читателя книга поможет ему быстро научиться заниматься математикой с помощью системы „Математика“. Основное внимание мы уделяем приемам выполнения базовых математических преобразований и программированию. Используя метафору, можно сказать, что это книга по вождению, а не устройству „автомобиля“. Опытные пользователи „Математики“, а также специалисты по программированию и информатике, которых в большей степени интересует именно устройство, найдут для себя полезной монографию В.З. Аладьева, Ю.Я. Хунта, М.Л. Шишакова [1]. Наша книга предназначается для новичков в символьных вычислениях и ни в коей мере не может заменить фундаментального руководства для пользователей С. Вольфрама [2]. При написании книги автор стремился обобщить опыт своей многолетней работы с продуктом, чтения основанных на „Математике“ курсов лекций в Московском институте электроники и математики (МИЭМ) и Московском государственном университете экономики, статистики и информатики (МЭСИ), а также участия в бета-тестировании ее новой версии. Изложение базируется на „Математике“ версии 2.2. Технические приемы работы с этим программным продуктом лишь в малой степени зависят от компьютерной платформы, однако первая глава книги, в которой

описывается интерфейс продукта, ориентирована в основном на „Математику“ под Windows. Помимо статей из журнала „Mathematica“ Journal и материалов электронного архива фирмы укажем на книги Дж. Грэя [3], Р. Мэдера [4], [5], Д. Введенского [6], Р. Гилорда, С. Камина и П. Веллина [7], откуда мы почерпнули идеи ряда примеров и упражнений. Просим ваши отзывы и предложения направлять по адресу *emv@amath.msk.ru.*

Выражаю свою искреннюю признательность профессору Питеру Дж. Олверу за впечатляющую демонстрацию эффективности „Математики“ для символьных вычислений по симметриям дифференциальных уравнений, студентам МИЭМ Олегу Гришину, Антону Джамаю, Михаилу Фурсову, старшему преподавателю В.Н. Жихареву, участвовавшим в написании специализированного пакета символьных симметрийных вычислений SYMMAN ([8]). Автор признателен А. Бочарову, С. Вольфраму и Л. Снятицкому за гостеприимство и интерес, проявленный ими к научным результатам автора во время его стажировки на фирме Wolfram Research, Inc. в рамках программы исследовательских грантов.

ЧАСТЬ I

„МАТЕМАТИКА“ КАК СИМВОЛЬНЫЙ, ГРАФИЧЕСКИЙ И ЧИСЛЕННЫЙ КАЛЬКУЛЯТОР

Первый шаг в овладении „Математикой“ — налаживание диалога с ней, хотя настоящее мастерство демонстрируют те пользователи, *для* которых „Математика“ является в первую очередь языком программирования высокого уровня. Диалоговый способ использования „Математики“ предполагает ее трактовку как большого калькулятора. Именно так она понимается в первой части книги. Для большой категории пользователей, заинтересованных в проведении прикладных расчетов, этого будет, по-видимому, вполне достаточно.

Глава 1
АЗБУКА „МАТЕМАТИКИ“

Предполагается, что изучение книги читателем происходит у персонального компьютера, на котором инсталлирована „Математика“. Особенно полезны параллельные с чтением вычисления именно для первой главы, в которой описываются основные приемы работы с программным продуктом и дается обзор его возможностей.

1.1. Первый сеанс

„Математика“, подобно любой другой прикладной программе, помещенной в оболочку Windows, начинает работать после того, как вы дважды щелкнете левой кнопкой мышки при положении ее указателя, направленном на пиктограмму „Математики“. Открывается прямоугольное рабочее окно, в котором имеется горизонтальная черная линия в верхней части экрана. Курсор в окне появится как только с клавиатуры будет введен любой знак, в том числе пробел. В левом верхнем углу экрана будут находиться введенный знак и справа от него курсор. Кроме того, в правом верхнем углу экрана появится синяя квадратная скобка, очерчивающая вертикальные границы ячейки, содержащей введенный с клавиатуры символ.

Напечатаем простейшее арифметическое выражение: $2+3$. Нажатие клавиш **Shift+Enter**, или **Insert**, или клавиши **5** в цифровой части клавиатуры при положении курсора внутри ячейки заставит „Математику“ вычислить введенное выражение. После вычисления на экране можно увидеть следующее:

In[1] := 2 + 3
Out[1] = 5

Кроме того, под вычисленным выражением имеется горизонтальная черта, ниже которой слева появится первый знак нового выражения, как только вы его начнете впечатывать. Исходное выражение присвоено в качестве значения объекту In[1], а результат - объекту Out[1]. Эти объекты появляются после вычисления, поэтому не следует ожидать In[2]:= как приглашения программы к вводу следующего выражения. В дальнейшем ввод с клавиатуры In[1] эквивалентен впечатыванию $2+3$, а ввод Out[1] равносилен вводу 5. Знаки := и = имеют существенно различный смысл и называются *отложенным* и *немедленным* присвоением. С помощью := присваивается в качестве значения невычисленное, а с помощью = вычисленное выражения.

Результат вычисления помещен в новую ячейку на экране, ограничивающая скобка которой отличается от скобки входной ячейки. Это означает, что входные и выходные ячейки имеют разные свойства. В любую входную ячейку можно переместить курсор, отредактировать ее, заменив часть или все входное выражение, и затем заново вычислить. В выходные ячейки нельзя переместить курсор без выполнения некоторых предварительных операций, о которых мы скажем позже. Кроме того, и входную, и выходную ячейки окаймляет еще одна скобка. Ее можно использовать для того, чтобы „спрятать“ выходную ячейку. Это может потребоваться, например, в случае, когда вычисленное выражение громоздко, поэтому выходная ячейка занимает слишком много места на экране.

Для того чтобы временно закрыть выходную ячейку, подводят указатель мышки к внешней скобке справа. Указатель при этом принимает вид направленной влево стрелки, ограниченной у ее острия вертикальным отрезком. Дважды щелкают левой кнопкой мышки, после чего на экране остается видна только входная ячейка. Окаймляющая скобка окрашивается в черный цвет, и горизонтальная линия исчезает. Без нее попытка ввести с клавиатуры какой-либо знак успеха не имеет, поэтому следует восстановить эту линию, щелкнув левой кнопкой

мышки при положении ее указателя ниже измененной ячейки. Открывают для обозрения спрятанную выходную ячейку тем же методом.

Продолжим наше знакомство с правилами ввода арифметических выражений. Произведение чисел 2 и 3 можно ввести либо в виде $2*3$, либо в виде разделенных пробелом цифр 2 и 3, т.е. как 2 3:

In[2] := 2 3
Out[2] = 6

С помощью наклонной черты / вводятся дроби:

In[3] := 105/335
Out[3] = $\frac{21}{67}$

а с помощью знака ^ — степени:

In[4] := 5^10
Out[4] = 9765625

При вводе более сложных выражений следует иметь в виду, что „Математика“ придерживается традиционных соглашений о старшинстве арифметических операций. Таким образом, выражение $2+3*4$ вычислится как 14, а не как $(2+3)*4$, т.е. 20. Тем не менее не следует экономить на круглых скобках (и), используемых для группирования. В „Математике“ их приходится использовать гораздо чаще, чем обычно, в силу „одноэтажного“ характера входного формата. Традиционно можно написать: $2^{3/4}5$, а в „Математике“ без круглых скобок не обойтись: 2^(3/4) 5. Если напечатать 2^3/4 5,то это будет воспринято как $(2^3/4)$ 5. Выражение $a/b/c$ будет автоматически сгруппировано как $(a/b)/c$; *a^b^c* как *a^(b^c)*; поэтому не стесняйтесь ставить круглые скобки, чтобы точно выразить, что вы хотите вычислить.

Кроме круглых скобок во входные выражения могут входить квадратные [,] и фигурные { , } скобки. Квадратные скобки используются для выделения аргументов функций или команд:

In[5] := Log[35.1]
Out[5] = 3.5582

Мы вычислили натуральный логарифм вещественного числа 35.1. Отметим, что вещественные числа записываются с точкой в качестве разделителя целой и дробной части. Кроме того, уместно заметить, что строчные и прописные буквы различаются и поэтому имя **Log** логарифмической функции нельзя ввести как **LOG** или как **log**.

„Математика" содержит около тысячи встроенных функций (команд). Поэтому ее использование как символьного, графического и численного калькулятора сводится в основном к применению различных функций к исходным выражениям, затем к функциям от исходных выражений, затем к функциям от функций от исходных выражений и т.д.

In[6] := Factor[x^5 - 2^5]
Out[6] = $(-2+x)(16+8x+4x^2+2x^3+x^4)$
In[7] := Last[Out[6]]
Out[7] = $16+8x+4x^2+2x^3+x^4$

Функция **Factor** раскладывает полиномы на множители, а **Last** в данном случае имеет значением последний множитель произведения. Кстати, выражение **In[7]** можно было бы ввести в виде **Last[%6]**, что более удобно.

Для ввода длинных выражений может оказаться мало одной строчки ячейки. В этом случае нажатием клавиши **Enter** можно перейти на новую строчку той же ячейки. При этом в конце строки будет автоматически поставлен пробел, который в некоторых случаях может быть воспринят как знак умножения.

Чтобы избежать этого, ставят знак \ в конце строки перед переносом. Если этого не сделать, то нужно соблюсти следующее правило: оставшиеся в первой строке буквы, цифры и другие знаки не должны порождать самостоятельное, целостное выражение „Математики“. В противном случае это выражение будет вычислено отдельно. Иначе скажем, что в ячейку можно впечатать с клавиатуры несколько выражений, начинающихся каждое с новой строки, которые по отдельности и последовательно будут вычислены.

Если же вводится одно выражение, занимающее несколько строк ячейки, то либо все выражение следует заключить в круглые скобки, либо каждую строчку следует заканчивать так, чтобы было ясно, что выражение вводом не закончено. Например, можно оканчивать строчку знаками +, −, / или открывающей квадратной скобкой и т.п.

Фигурные скобки в „Математике“ используются для ввода и вывода списков: $\{a,b,c\}$, $\{x,2,Abs[z],y\hat{}2\}$ и т.д. Элементами списка могут быть любые выражения, в том числе и списки. Список $\{\{a1,a2,a3\},\{b1,b2,b3\},\{c1,c2,c3\}\}$ можно понимать как матрицу 3×3, записанную построчно:

In[8] := m = {{a1,a2,a3},{b1,b2,b3},{c1,c2,c3}}

Out[8] = {{*a*1,*a*2,*a*3},{*b*1,*b*2,*b*3},{*c*1,*c*2,*c*3}}

In[9] := m//MatrixForm

$$Out[9]//MatrixForm = \begin{matrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{matrix}$$

Если в предыдущих примерах символ функции предшествовал аргументу, т.е. использовалась префиксная форма записи, то в последнем примере функция одного аргумента **MatrixForm** поставлена после своего аргумента, или, иначе, записана в постфиксной форме. Кроме того, результат восьмого вычи-

сления присвоен в качестве значения не только объекту Out[8], но и символу m, что гораздо удобнее со многих точек зрения и является общеупотребительным приемом вычислений.

При вводе выражений могут произойти ошибки, или возникнет необходимость изменить или скопировать часть выражения в какой-либо входной ячейке. В этом случае следует прибегнуть к редактированию. Для входных ячеек это можно сделать стандартными приемами, принятыми в текстовых редакторах, работающих под Windows, т.е. с помощью Clipboard и команд **Edit Cut**, **Edit Copy**, **Edit Paste**.

Удалить, скопировать, вставить можно не только любую часть содержимого входной ячейки, но и всю ячейку в целом. Для этого ее следует *выделить*, подведя указатель мышки справа к скобке ячейки и щелкнув левой клавишей мышки. Скобка изменит цвет на черный. С помощью Clipboard ячейку или ее часть можно вставить в любой другой документ прикладной Windows программы. Редактировать можно и выходные ячейки с результатами вычислений. Естественно, что можно удалить или скопировать всю выходную ячейку. Если же требуется изменить часть выходной ячейки, то ее следует предварительно сделать неформатированной. Для этого выходная ячейка выделяется, и в меню **Cell** выбирается команда **Formatted**. После ее выполнения содержимое выходной ячейки будет записано во входном формате, что позволит изменить, вырезать или скопировать часть ячейки. После редактирования выражение в рассматриваемой ячейке можно вычислить, для чего ее делают активной с помощью **Cell Inactive**. Таким образом, после описанных действий бывшая выходная ячейка становится входной.

Работа с „Математикой" оформляется в виде сохраняемого в отдельном файле интерактивного электронного документа, называемого Записной книжкой. Записные книжки могут содержать в своих ячейках обычный текст, вычисляемые выражения, результаты вычислений и графику. Если вы хотите дать название Записной книжке, то напечатайте это название

в отдельной ячейке в начале документа. При условии, что ваш компьютер русифицирован, можно переключить клавиатуру на русские буквы, хотя, возможно, в этот момент выводимые на экран символы не будут выглядеть как русские буквы. Выделите ячейку, обратитесь к меню **Style**, находящемуся в верхней части рабочего окна, и выберите команду **Cell Style**. В меню появится перечисление всех типов ячеек, которые могут содержаться в записных книжках. По умолчанию, любая новая ячейка имеет тип **Input**, т.е. является неформатированной и активной. Выберите тип **Title** или более скромный **Subtitle**. Вы увидите, что текст в ячейке изменил размер. Снова обратитесь к меню **Style** и выберите пункт меню **Font**. В появившемся окне диалога будет содержаться информация обо всех имеющихся типах шрифтов, об их размерах и цвете. Если оболочка **Windows** снабжена кириллическими шрифтами, то ими же оснащается и „Математика“. Выберите шрифт, размер и цвет, нажмите OK, и Записная книжка озаглавлена. При условии, что произошло переключение клавиатуры и выбран кириллический шрифт, название книжки появится на русском языке. Полезно перемежать вычисления комментариями. Комментарии лучше всего помещать в ячейки, имеющие тип **Text**.

При необходимости в конце сеанса работы с „Математикой“ Записная книжка сохраняется. Это делается стандартно с помощью **File Save As**. Имя файла не обязано совпадать с заглавием записной книжки. Во время сеанса можно открыть несколько Записных книжек, прибегнув к **File New**.

1.2. Основы синтаксиса „Математики“

Два фундаментальных понятия лежат в основе работы „Математики“: *выражение* и *вычисление*. Выражения — основной тип данных в „Математике“. Определим это понятие рекурсивно, начиная с атомарных выражений: символов, чисел и строк.

Символы есть основные именованные объекты. Имя символа — это любая последовательность латинских строчных или прописных букв, цифр и знака $, не начинающаяся с цифры. Строчные и прописные буквы различаются. Например, x, aB, $u\$v25$ и т.д. есть символы.

Числа в „Математике“ рассматриваются четырех типов: целые, рациональные, вещественные и комплексные. Все типы чисел могут содержать любое количество цифр. Чтобы число рассматривалось как вещественное, оно должно содержать точку в его записи, даже в случае, если имеет нулевую мантиссу. При вычислениях „Математика“ сохраняет, если это возможно, тип чисел. Например, квадратный корень из 4, представленный выражением **Sqrt[4]**, вычислится как 2, а квадратный корень из 2, т.е. **Sqrt[2]**, после вычисления будет также записан в виде **Sqrt[2]**. Для того чтобы получить приближенное значение этого корня, следует вычислить квадратный корень **Sqrt[2.]** из вещественного числа 2, представленного как 2. или как 2.0 в его записи.

Строки — это заключенные в кавычки последовательности букв, цифр и специальных символов: „This is a string“. Если внутри строки используются кавычки, то их следует представить в виде последовательности \". Строки также могут содержать последовательности: $\backslash n$ — перехода на новую строчку, $\backslash t$ — табуляции и $\backslash n_1 n_2 n_3$, где $n_1 n_2 n_3$ — восьмеричный код ASCII.

Произвольные выражения „Математики“ строятся, начиная с атомарных, и имеют вид: **h[e1,e2,...,en]**, где h называется заголовком выражения, а e_1, e_2, ... e_n — элементами, или непосредственными подвыражениями, или частями рассматриваемого выражения. И заголовок, и элементы также являются выражениями.

Если заголовок — символ, то возможны следующие основные интерпретации его смысла. Заголовок можно понимать как имя математической функции: **Sin** (синус), **Exp** (экспоненциальная функция), как команду: **Factor** (разложить на

множители), **Delete** (удалить), как тип данных: **Integer** (целое), **List** (список). Мы привели в качестве примеров встроенные (системные) заголовки. Все такие заголовки начинаются с заглавной латинской буквы и являются полными именами на английском соответствующего объекта. Несмотря на то что заголовки совсем не обязательно интерпретируются как имена функций, мы далее будем называть их функциями, следуя традиции, установившейся в литературе по „Математике“.

Сказанное выше относительно выражений „Математики“ справедливо относительно внутренней, или полной формы выражений, т.е. той формы, которая воспринимается вычислительным ядром системы. Синтаксис выражений, впечатываемых с клавиатуры во входные ячейки, может отличаться от синтаксиса полной формы.

Рассмотрим следующие выражения:

Входной формат выражения	Внутренняя (полная) форма выражения	Выходной формат выражения
x+a	Plus[a, x]	a+x
x a	Times[a,x]	a x
x*a	Times[a,x]	a x
x^a	Power[x,a]	x^a
x–a	Plus[Times[-1,a], x]	–a+x
x/a	Times[Power[a,-1], x]	$\frac{x}{a}$

Мы видим, что существуют три формата выражений. Входной формат характеризуется тем, что выражение печатается в одну строку, и общеупотребительные математические операции могут быть записаны в специальной, инфиксной форме. Более того, в инфиксной форме могут быть записаны во входном формате любые функции „Математики“ от двух аргументов: $\{a, b\} \sim$ **Delete** ~ 1. Мы уже упоминали, что функции одного аргумента могут быть записаны и в постфиксной форме, т.е. после своего аргумента. Выходной формат максимально приближен к традиционному представлению математических выражений и может потребовать несколько строк для выво-

да выражения. Наконец, вторая колонка дает представление математических формул в виде выражений „Математики“. В рассмотренных примерах вычисления свелись к сортировке, т.е. к упорядочиванию символов исходного выражения в соответствии с встроенным в „Математику“ порядком.

Функция **AtomQ**, примененная к аргументу *expr*, т.е. выражение **AtomQ**[*expr*] при вычислении имеет результат **True**, если выражение *expr*, *будучи вычисленным*, является атомарным, и **False** — в противном случае. Заголовок вычисленного выражения *expr* можно найти, вычисляя **Head**[*expr*]. Например, **Head**[**2**] есть **Integer**, **Head**[**b$x**] есть **Symbol**, **Head**[**„I read a book“**] есть **String**. Наконец, элемент e_k можно получить с помощью функции **Part** путем вычисления выражения **Part**[*expr*, **k**], или, что то же самое, выражения **expr**[[*k*]]. Если выражение атомарное, то из него нельзя извлечь никакой части.

О том, как вычисляются выражения, мы будем подробно говорить в гл. 8. Здесь лишь упомянем, что вычисления основываются на правилах преобразования выражений, правил как встроенных, или системных, так и определяемых пользователем. Выражение считается вычисленным, если оно не изменяется при применении к нему любого из подобных правил преобразований.

Проследить за ходом вычислений можно с помощью функции **Trace** или **TracePrint**.

In[**10**] := **Trace**[**2**(**3^2** + **1**)]

Out[10] = $\{\{\{3^2, 9\}, 9+1, 1+9, 10\}, 2\ 10, 20\}$

Пример показывает, что „Математика“ следует обычным правилам вычисления, которые применяются для того, чтобы вычислить арифметическое выражение. Сначала вычисляется степень, потом круглая скобка, потом произведение. Функции **Trace** и **TracePrint** имеют вторые аргументы, которые

позволяют следить не за всеми, а за ключевыми, с вашей точки зрения, этапами вычисления. Об этих аргументах можно узнать, напечатав на экране ? **Trace**. Ясно, что эта форма получения информации о функции **Trace** носит общий характер и применяется ко всем встроенным функциям „Математики“. Для получения более полной информации вместо одного знака ? следует напечатать ??.

1.3. Обзор „Математики“

Чтобы читатель мог получить самое общее, пусть и поверхностное, представление о возможностях программного продукта, приведем несколько характерных примеров. Попутно мы познакомимся с некоторыми из наиболее часто используемых функций „Математики“. Начиная с этого момента в записях диалога с компьютером мы не будем печатать объекты In[] и Out[], как излишние. Входные ячейки, т.е. ячейки, содержимое которых вводится пользователем, будут по-прежнему печататься полужирным шрифтом, а выходные — обычным. Это позволит легко распознавать, что предназначено для вычисления и что вычислено ядром „Математики“.

АЛГЕБРА. Пусть требуется найти корни уравнения третьей степени $x^3 + ax + b = 0$ с символьными коэффициентами a и b. Как известно, имеются общие формулы для корней полиномиальных уравнений степени не выше четвертой. Воспользуемся функцией **Solve** для того, чтобы найти корни рассматриваемого уравнения. Для записи в уравнениях отношения равенства в „Математике“ используется знак ==, являющийся инфиксной формой функции **Equal**. Первым аргументом функции **Solve** будет являться рассматриваемое уравнение, написанное с помощью ==, а вторым аргументом — неизвестная, относительно которой решается уравнение. Последнее необходимо, так как в

уравнениях, кроме символов неизвестных, могут содержаться и другие символы. В рассматриваемое уравнение наряду с x входят символы a и b, которые также могли бы трактоваться как неизвестные.

Solve[x^3 + a x + b == 0, x]

$$\{\{x \to -\frac{2^{1/3}\,a}{(-27b + Sqrt[108a^3 + 729b^2])^{1/3}} + \frac{(-27b + Sqrt[108a^3 + 729b^2])^{1/3}}{3\ 2^{1/3}}\},$$

$$\{x \to \frac{(1 + I\,Sqrt[3])a}{2^{2/3}(-27b + Sqrt[108a^3 + 729b^2])^{1/3}} - \frac{(1 - I\,Sqrt[3])(-27b + Sqrt[108a^3 + 729b^2])^{1/3}}{6\ 2^{1/3}}\},$$

$$\{x \to \frac{(1 - I\,Sqrt[3])a}{2^{2/3}(-27b + Sqrt[108a^3 + 729b^2])^{1/3}} - \frac{(1 + I\,Sqrt[3])(-27b + Sqrt[108a^3 + 729b^2])^{1/3}}{6\ 2^{1/3}}\}\}$$

Ответ представляет собой список из трех элементов, соответствующих трем корням уравнения. Каждый из этих элементов также является списком, содержащим один элемент. Корни уравнения представлены в виде правил подстановок $x \to expr$, что упрощает, как мы увидим ниже, дальнейшие манипуляции с корнями, в частности, вычисление различных выражений, содержащих x. Знак $\to$, являющийся инфиксной формой функции **Rule**, при необходимости вводится с клавиатуры с помощью последовательно набираемых знаков - (дефис) и > (строго больше). В ответе содержится помимо знакомой нам функции **Sqrt**, извлекающей квадратный корень из своего аргумента, мнимая единица I, которая, как и всякий системный символ, напечатана с заглавной буквы.

Можно попытаться найти в явном виде корни некоторых полиномиальных уравнений степени выше четвертой.

Solve[x^5 + x − 1 == 0, x]

$$\{\{x \to \frac{1 - I\,Sqrt[3]}{2}\}, \{x \to \frac{1 + I\,Sqrt[3]}{2}\},$$

$$\{x \to -\frac{1}{3} + \frac{2^{1/3}}{3(25 + 3\,Sqrt[69])^{1/3}} + \frac{(25 + 3\,Sqrt[69])^{1/3}}{3\;2^{1/3}}\},$$

$$\{x \to -\frac{1}{3} - \frac{1 + I\,Sqrt[3]}{3\;2^{1/3}(25 + 3\,Sqrt[69])^{1/3}} - \frac{(1 - I\,Sqrt[3])(25 + 3\,Sqrt[69])^{1/3}}{6\;2^{1/3}}\},$$

$$\{x \to -\frac{1}{3} - \frac{1 - I\,Sqrt[3]}{3\;2^{1/3}(25 + 3\,Sqrt[69])^{1/3}} - \frac{(1 + I\,Sqrt[3])(25 + 3\,Sqrt[69])^{1/3}}{6\;2^{1/3}}\}\}$$

Поскольку уравнение не содержит символьных параметров, то естественно поинтересоваться приближенными значениями корней. С этой целью в самом уравнении можно было бы записать единицу в формате вещественного числа, и тогда был бы получен приближенный ответ. Можно также воспользоваться общим рецептом получения приближенных значений числовых выражений, применив к результату функцию **N**, преобразующую все числа в вещественные. Функция **N** является исключением из общего правила записи системных функций в виде полных английских слов и является, по-видимому, первой буквой слова Numeric. Применим функцию **N** к полученному результату в постфиксной форме. В „Математике“ только что полученный результат присваивается системному символу %, поэтому функцию **N** можно вычислить от аргумента %.

%//N

$$\{\{x \to 0.5 - 0.866025\,I\}, \{x \to 0.5 + 0.866025\,I\},$$

$$\{x \to 0.754878\}, \{x \to -0.877439 + 0.744862\,I\},$$

$$\{x \to -0.877439 - 0.744862\,I\}\}$$

Попробуем теперь найти корни уравнения $x^5 + 2x - 1 = 0$.

rts = Solve[x^5 + 2x - 1 == 0, x]

$\{ToRules[Roots[2x + x^5 == 1, x]]\}$

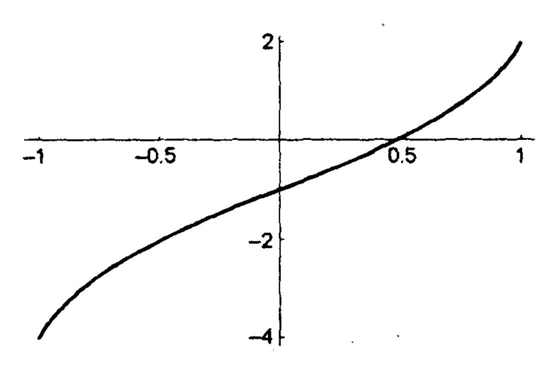
Подобный ответ означает, что формул для корней рассматриваемого уравнения не существует. Тем не менее можно узнать численное значение всех пяти корней, вычислив выражение **N[rts]**.

N[rts]

$\{\{x \to -0.945068 - 0.854518\,I\}, \{x \to -0.945068 + 0.854518\,I\},$
$\{x \to 0.486389\}, \{x \to 0.701874 - 0.879697\,I\},$
$\{x \to 0.701874 + 0.879697\,I\}\}$

Можно убедиться в правильности вычисления хотя бы одного вещественного корня, нарисовав график многочлена $x^5 + {} + 2x - 1$ с помощью графической функции **Plot** (рис. 1.1):

Plot[x^5 + 2x - 1, {x, -1, 1}]



– Graphics –

Рис. 1.1

Действительно, график рассматриваемого многочлена пересекает ось Ox вблизи от точки $x = 0.5$. Сообщение *-Graphics-* означает, что „Математика“ сформировала описание представленного графика на языке PostScript. Вот фрагмент этого описания:

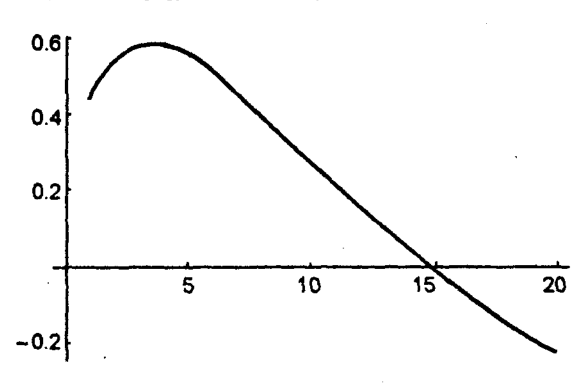
%!
%% Creator: Mathematica
%%AspectRatio: .6103
MathPicture Start
%%Graphics
/ Courier findfont 10 scalefont setfont
%Scaling calculations
0.5 0.47619 0.407118 0.0981006 [
[(-1)] .02381 .40712 0 2 *Msboxa*

Рассматриваемый текст можно увидеть и при необходимости отредактировать, расформатировав ячейку, в которой содержится график. Для этого ячейку выделяют и выполняют **Cell Formatted**. Этот и ему подобные тексты можно преобразовать в различные графические форматы: EPS, PICT, PCX и другие, сохранить в файле, использовать для вставки в текст статьи, чтобы послать коллегам по электронной почте и т.п.

Возвращаясь к решению уравнений, заметим, что „Математика“ умеет численно решать не только полиномиальные, но и трансцендентные уравнения.

Правда, если заранее известно, что какое-то уравнение имеет бесконечно много корней, следует указать хотя бы грубое приближение к искомому корню. Последнее можно найти, нарисовав график соответствующей функции. Предположим, что нам нужно найти первый после нуля корень функции Бесселя первого порядка J_1 аргумента $\sqrt{x}$, т.е. корень функции $J_1(\sqrt{x})$ (рис. 1.2).

Plot[BesselJ[1,Sqrt[x]],{x,1,20}]



– *Graphics* –

Рис. 1.2

Мы видим, что в рассматриваемом интервале корень уравнения существует и лежит вблизи точки $x = 15$. Уточним его значение.

FindRoot[BesselJ[1,Sqrt[x]],{x,15}]
$\{x \to 14.682\}$

МАТЕМАТИЧЕСКИЙ АНАЛИЗ. Наряду с алгебраическими преобразованиями „Математика“ позволяет выполнять операции математического анализа. Естественно, что базовыми являются операции интегрирования **Integrate** и дифференцирования **D** (вновь исключение, впрочем, последнее из правила):

int = Integrate[x^5/Sqrt[x^3 – 1],x]

$$\left(\frac{4}{9} + \frac{2x^3}{9}\right) Sqrt[-1 + x^3]$$

Проверим правильность полученного ответа дифференцированием:

difint = D[int,x]

$$\frac{3x^2\left(\frac{4}{9}+\frac{2x^3}{9}\right)}{2\,Sqrt[-1+x^3]}+\frac{2x^2\,Sqrt[-1+x^3]}{3}$$

Полученный результат не совпадает с исходным выражением. Попробуем его упростить:

difint//Simplify

$$\frac{x^5}{Sqrt[-1+x^3]}$$

Одной из важнейших математических задач является нахождение решений обыкновенных дифференциальных уравнений. Для этой цели предназначена функция **DSolve**, первым аргументом которой является дифференциальное уравнение или система дифференциальных уравнений. Если $y[x]$ есть неизвестная функция, то производная от нее может быть записана во входном формате как $y'[x]$. Вторым аргументом функции **DSolve** является неизвестная функция, а третьим — независимая переменная.

DSolve[x^2 y''[x] + x y'[x] + x^2 y[x] == 0, y[x], x]

$$\{\{y(x) \to BesselY[0,x]C[1] + BesselJ[0,x]C[2]\}\}$$

Получено общее решение исследуемого дифференциального уравнения, являющееся линейной комбинацией с коэффициентами $C[1]$ и $C[2]$ функций Бесселя нулевого порядка первого и второго рода.

Функцию **DSolve** можно применять и для решения систем обыкновенных дифференциальных уравнений.

**DSolve[{y'[x] == a^2z[x] - Sin[x], z'[x] == y[x] + b}
{y[x], z[x]}, x],**

$$\{\{y[x] \to -\frac{b}{1+a^2} - \frac{a^2 b}{1+a^2} - \frac{aC[1]}{E^{ax}} + aE^{ax}C[2] + \frac{Cos[x]}{1+a^2},$$

$$z[x] \to \frac{C[1]}{E^{ax}} + E^{ax}C[2] + \frac{Sin[x]}{1+a^2}\}\}$$

Не все обыкновенные дифференциальные уравнения и системы уравнений допускают явные точные решения, поэтому приходится прибегать к численным методам. Обратимся к системе обыкновенных дифференциальных уравнений, описывающих плоское движение двух гравитирующих тел. Пусть масса второго тела в 1,3 раза больше массы первого. В этом случае движение описывается следующей системой обыкновенных дифференциальных уравнений:

```
masses = {x1''[t] == -1.3(x1[t] - x2[t]) /
((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2)^(3/2),
y1''[t] == -1.3(y1[t] - y2[t]) /
((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2)^(3/2),
x2''[t] == -(x2[t] - x1[t]) /
((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2)^(3/2),
y2''[t] == -(y2[t] - y1[t]) /
((x1[t] - x2[t])^2 + (y1[t] - y2[t])^2)^(3/2)};
```

Знак пунктуации ; (точка с запятой), поставленный в конце входного выражения, блокирует вывод на экран результата вычисления, хотя рассматриваемая система присвоена в качестве значения символу *masses*. Снабдим систему *masses* следующими начальными данными:

```
init =
{x1[0] == 1, x1'[0] == 0, y1[0] == 0, y1'[0] == 0.3,
x2[0] == -10/13, x2'[0] == 0, y2[0] == 0, y2'[0] == -3/13};
```

и найдем численное решение поставленной задачи Коши на интервале $0 \leq t \leq 4$.

Для численного решения задач Коши для систем обыкновенных дифференциальных уравнений предназначена функция **NDSolve**. Ее первый аргумент — исследуемая система, дополненная начальными условиями; второй аргумент — список имен неизвестных функций; третий аргумент — список, содержащий независимую переменную, начальную и конечную точки интервала численного интегрирования системы.

```
n = NDSolve[Join[masses, init], {x1, y1, x2, y2}, {t, 0, 4}]
{ {x1 → InterpolatingFunction[{0., 4.}, <>],
y1 → InterpolatingFunction[{0., 4.}, <>],
x2 → InterpolatingFunction[{0., 4.}, <>],
y2 → InterpolatingFunction[{0., 4.}, <>], } }
```

Результатом вычисления являются четыре интерполяционные функции, используя которые можно нарисовать орбиты тел (рис. 1.3):

```
ParametricPlot[{Evaluate[{x1[t], y1[t]}, /.n],
Evaluate[{x2[t], y2[t]}/.n]}, {t, 0, 4},
AspectRatio → Automatic];
```

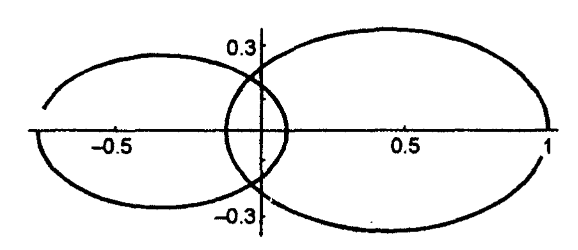


Рис. 1.3

Начальные данные подобраны так, что гравитирующие тела описывают замкнутые эллиптические орбиты. Поскольку вычисления занимают мало времени, читатель может начать

изучение закономерностей движения тел, изменяя числовые параметры в *init* или *masses*.

„Математику“ можно использовать как справочник специальных математических функций: Бесселя, Лежандра, Эйлера, Эйри, Эрмита и т.д. Сведения об асимптотических разложениях функции Бесселя первого рода второго порядка в окрестностях точек 0 и ∞ можно получить, используя функцию **Series**, следующим образом:

{Series[BesselJ[2,x],{x,0,2}],Series[BesselJ[2,x],
{x,Infinity,2}]}

$$\{\frac{x^2}{8}+O[x]^4,$$
$$Sqrt[2]Sqrt[\frac{1}{Pix}]\Big(Cos[\frac{5Pi}{4}-x](1-\frac{105}{128x^2}+O[\frac{1}{x}]^3)$$
$$+(\frac{15}{8x}+O[\frac{1}{x}]^3)Sin[\frac{5Pi}{4}-x]\Big)\}$$

В полученной формуле символ *Pi* есть число π. С помощью функции Series можно раскладывать в асимптотические ряды произвольные функции:

Series[(f[x+h]-2f[x]+f[x-h])/h^2,{h,0,3}]

$$f''[x]+\frac{f^{(4)}[x]h^2}{12}+O[h]^4$$

АППРОКСИМАЦИЯ. Рассмотрим задачу о наилучшем среднеквадратичном приближении. Предположим, что в файле *file.val*, расположенном в вашей рабочей директории, хранятся экспериментальные данные. Содержимое файла можно увидеть, выполнив команду:

!!file.val

1	−0.75
0.2	0.81077
0.4	1.31766

и т.д. (мы не приводим всех строк файла в целях экономии места). В файле содержатся пары вещественных чисел, записанные построчно и представляющие значения абсцисс и ординат экспериментальных данных. Файл можно ввести в Записную книжку с помощью команды **ReadList**, присвоив символу *data* значение соответствующего списка:

data = ReadList["file.val", {Number, Number}]
{{0, –0.75}, {0.2, 0.81077}, {0.4, 1.31766}, {0.61, 1.63181},
{0.8, 2.2533}, {1., 1.75}, {1.2, 1.23509}, {1.4, 1.45863},
{1.6, 1.293237}, {1.8, 0.914413}, {2., 1.245}, {2.2, 1.82007},
{2.4, 1.72712}, {2.6, 2.21069}, {2.8, 3.719648}, {3., 3.75} }

Представим данные *data* в виде дискретного графика (рис. 1.4):

**p1 = ListPlot[data, PlotStyle → {PointSize[0.01],
RGBColor[1, 0, 0]}];**

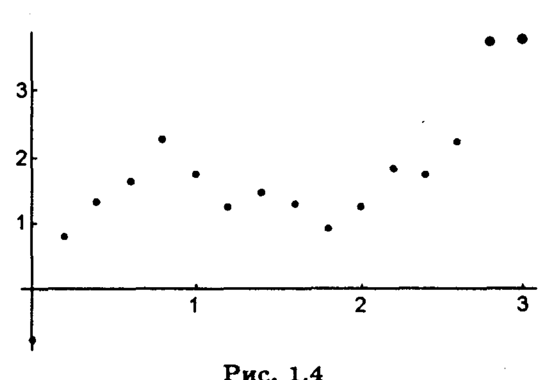


Рис. 1.4

Попробуем аппроксимировать дискретные данные data на отрезке от 0 до 3 с помощью полинома четвертой степени:

poly4 = Fit[data, {1, x, x^2, x^3, x^4}, x]
$-0.728077 + 8.77373x - 9.6255x^2 + 3.78372x^3 - 0.459172x^4$

Нарисуем график этого полинома на отрезке $0 \le x \le 3$ (рис. 1.5):

p2 = Plot[poly4, {x, 0, 3}, PlotStyle → {{Thickness[0.007], RGBColor[1, 0, 1]}}];

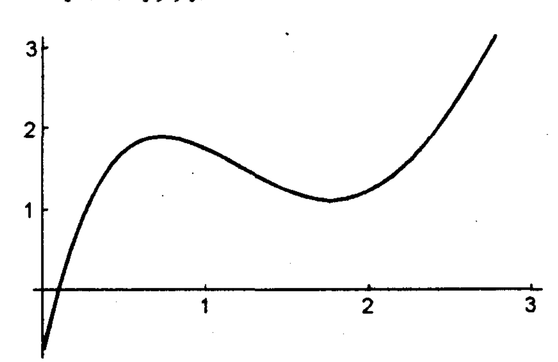


Рис. 1.5

С помощью функции **Show** совместим два последних графика (рис. 1.6). Если точность приближения неудовлетворительна, можно взять многочлен более высокой степени.

Show[p1, p2];

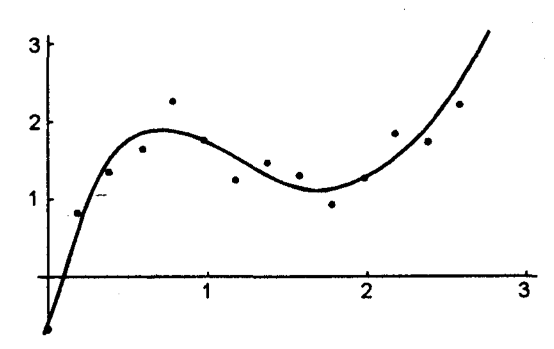


Рис. 1.6

ВИЗУАЛИЗАЦИЯ. С помощью „Математики“ можно получать не только двумерные, но и трехмерные графики (рис. 1.7):

```
Plot3D[AiryAiPrime[(x + y^2)/2]Sin[9 x y],
{x,0,Pi/2},{y,0,Pi/2}];
```

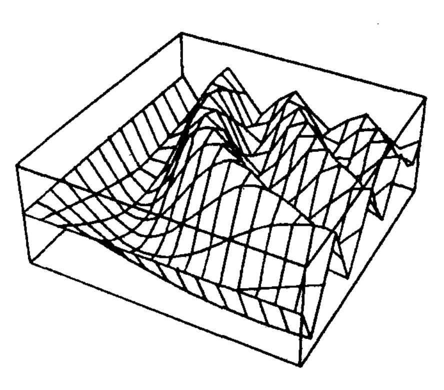


Рис. 1.7

Кроме того, функции от двух аргументов можно изучать с помощью функций **ContourPlot** и **DensityPlot**, рисующих контурные и плотностные графики (рис. 1.8):

```
ContourPlot[AiryAiPrime[(x + y^2)/2]Sin[9 x y],
{x,0,Pi/2},{y,0,Pi/2},PlotPoints → 25];
```

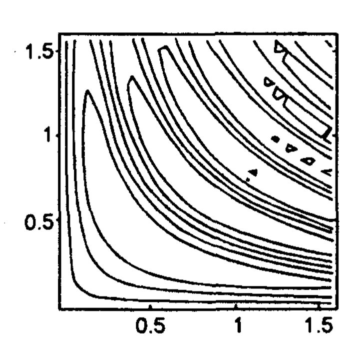


Рис. 1.8

В рассмотренных примерах первым аргументом графических функций является произведение производной от функции Эйри первого рода **AiryAiPrime** и функции **Sin** от полиномиальных аргументов, содержащих переменные x и y. Функции **Plot**, **Plot3D**, **ContourPlot** и **DensityPlot** не исчерпывают всего списка графических функций, которых более десятка. Все такие функции содержат средства для преобразования стиля рисунков с целью придания им большей выразительности и информативности. Выше мы уже воспользовались одним из этих средств: PlotStyle в функции **ListPlot**, задавая относительные размеры точек и их цвет.

ПРОГРАММИРОВАНИЕ. „Математика“ позволяет писать программы как в традиционном процедурном стиле языков Фортран, Паскаль и т.п., так и в функциональном стиле и стиле определений (правил преобразований). Простой пример функционального стиля предоставляет программа вычисления среднего значения и дисперсии случайной последовательности чисел. В „Математике“ последовательность чисел записывается в виде списка. Например, $l = \{2,4,9\}$ есть последовательность чисел 2, 4, 9. Определим функцию **mean** соотношением

mean[x_List] := Apply[Plus,x]/Length[x]

Смысл входящих в это определение функций следующий. **Length** имеет значением длину списка, **Plus** дает сумму своих аргументов, а **Apply** заменяет заголовки функций. В данном случае заголовок **List** заменяется на **Plus**. Заголовок **List**, поставленный после аргумента $x_$, означает, что функция **mean** определена только на множестве списков. Вычислим среднее значение элементов списка l:

mean[l]

5

Теперь определим функцию **var**, вычисляющую дисперсию:

```
var[x_List] := mean[(x - mean[x])^2]
```

и вычислим дисперсию случайной последовательности десяти натуральных чисел, заключенных между 1 и 1000.

```
data = Table[Random[Integer, {1,1000}], {i,10}]
{335,272,528,907,87,300,497,817,183,388}

var[data]//N
62268.2
```

Примером программирования в стиле правил преобразований является программа, в которой определяется следующая функция **minimum**, обрабатывающая списки. Для списков, состоящих из целых чисел, функция имеет результатом минимальное число в списке. Если список состоит из символов, то вычисляется первый символ в лексикографическом порядке, встроенном в „Математику“. Другие списки никак не преобразуются.

```
minimum[l : {x__Integer}] := Min[l]
minimum[l : {x__Symbol}] := First[Sort[l]]
minimum[l_List] := l
```

При вычислении выражения **minimum[l]**, где l есть произвольный список, прежде всего проверяется, являются ли все элементы списка целыми числами или все символами. Если реализуется одна из этих возможностей, то рассматриваемое выражение преобразуется в соответствии с первым или вторым правилом преобразования. Если элементы списка не удовлетворяют критериям, то список не изменяется. Наконец, если аргумент функции **minimum** не является списком, то исходное выражение остается невычисленным.

Упражнения

1. *Вычислите* выражения **N[Pi,3]**, **N[Pi,6]**, **N[Pi,100]**. Что они означают? *Вычислите* число e, представленное в „Математике" символом E, с пятьюдесятью верными цифрами. *Узнайте*, сколько времени займет на вашем компьютере нахождение приближенного значения числа π с тысячью верных цифр, вычислив выражение

   N[Pi,1000]//Timing

2. Как узнать, что больше: e^π или π^e?
3. С помощью функции **Factor** *разложите* на множители полиномы $a^8 - x^8$, $a^2 - x^2 + 2ay + y^2$ и $a^2 + 2\sqrt{2}ax + 2x^2$. Вычислите **Expand[(a+ + $\sqrt{2}$x)^2]**. О чем говорит сравнение двух последних результатов?
4. С помощью встроенной функции **D** *вычислите* смешанную производную от функции $y\exp(x^2y^3)$ по x и y. Сколько времени займет вычисление частной производной десятого порядка от этой функции по переменной x на вашем компьютере?
5. С помощью функции **Integrate** *вычислите* определенный интеграл от функции $x\sin x$ от 0 до π. Постарайтесь узнать, как это сделать, получив информацию о функции **Integrate** с помощью **?Integrate**.
6. *Найдите* решения систем уравнений $x^2 + y^2 = 1$, $x - y = 1/2$ и $x^2 + y^2 = 1$, $x - y = 0.5$. *Установите* геометрический смысл решений, вычислив выражение

   ```
   Plot[{Sqrt[1 - x^2], -Sqrt[1 - x^2], x - 1/2}, {x, -1, 1},
   PlotStyle -> {{RGBColor[0,0,1]}, {RGBColor[0,1,0]},
   {RGBColor[1,0,0]}}, AspectRatio -> Automatic]
   ```

7. *Найдите* численное значение наибольшего корня уравнения $\sin x = 0.1x$. Предварительно постройте графики функций $\sin x$ и $0.1x$ с помощью функции **Plot**.
8. *Вычислите* выражения **True && False** и **True || False**. Инфиксными формами каких логических функций являются && и ||?
9. *Вычислите* пять первых членов разложения функции $\ln(1-x)\ln(1+x)$ в ряд Тейлора в окрестности точки $x = 0$. Натуральный логарифм представлен в „Математике" функцией **Log**.
10. *Установите*, какая функция натурального аргумента определена соотношением

    f[n_Integer] := Apply[Times, Range[n]].

Глава 2
СИМВОЛЬНЫЕ ВЫЧИСЛЕНИЯ

„Математика" позволяет автоматизировать практически все типы символьных вычислений, встречающиеся в теоретической и прикладной математике. Ниже мы приведем обзор основных функций, применяемых при проведении символьных преобразований.

2.1. Преобразования многочленов

Начнем с простейших примеров. Присвоим символу $a1$ значение следующего алгебраического выражения:

a1 = (x + y)^3 + (x + y)(y − 1)

$$(-1+y)(x+y)+(x+y)^3$$

Мы видим, что сама по себе „Математика" не стремится раскрыть произведения. Это делает функция **Expand**, раскрывающая произведения и положительные степени сумм.

a2 = Expand[a1]

$$-x+x^3-y+xy+3x^2y+y^2+3xy^2+y^3$$

У функции **Expand** может быть второй аргумент. При вычислении выражения **Expand[expr, pattern]** не раскрываются те элементы expr, которые не содержат членов, отвечающих шаблону *pattern*.

Expand[a1, y − 1]

$$-x-y+y(x+y)+(x+y)^3$$

Expand[a1, x + y]

$x^3 + x(-1+y) + 3x^2y + (-1+y)y + 3xy^2 + y^3$

Противоположной функции **Expand** по действию является функция **Factor**, которая раскладывает полиномы на множители над полем рациональных чисел.

Factor[a2]

$(x+y)(-1+x^2+y+2xy+y^2)$

Обе рассматриваемые функции имеют дополнительные, не обязательно указываемые аргументы, или опции. Если один из этих аргументов задать в виде **Trig→True**, то тригонометрические функции будут трактоваться как рациональные функции экспонент.

Expand[2Sin[x]^3, Trig → True]

$\frac{3Sin[x]}{2} - \frac{Sin[3x]}{2}$

Factor[Sin[3x], Trig → True]

$(1+2Cos[2x])Sin[x]$

Аналогичными **Factor** являются следующие функции. Вычисленное выражение **FactorList[poly]** представляет собой список множителей полинома *poly* вместе с показателями степеней, с которыми они входят в разложение *poly* на множители. Первый элемент списка есть общий численный множитель, а если такового отличного от единицы нет, то список начинается с $\{1,1\}$. Например, **FactorList[2 (1 – x^4) (1 – x)]** при вычислении приводит к следующему результату: $\{\{2,1\},\{-1+x,2\}, \{1+x,1\},\{1+x^2,1\}\}$.

Функция **FactorTerms** позволяет вынести общий числовой множитель в слагаемых своего аргумента *poly*; вычисление **FactorTerms**[*poly*, x] позволяет вынести общий множитель, не зависящий от x; **FactorTerms[poly, {x1, x2, ...}]** последовательно выделяет множители, не зависящие от каждого x_i. Вычисленное выражение **FactorTermsList[poly, {x1, x2, ...}]** дает

список множителей *poly*. Первый элемент в списке есть общий числовой множитель, второй — множитель, не зависящий ни от одного из x_i. Последующие элементы есть множители, не зависящие от как можно большего числа x_i.

Выражение **Coefficient[poly,form]** после вычисления имеет своим значением коэффициент при выражении *form* в полиноме *poly*. Возведем многочлен *a*2 в десятую степень. Поскольку результат будет довольно громоздким, воспользуемся точкой с запятой, поставленной в конец соответствующего выражения с тем, чтобы заблокировать вывод на экран результата вычисления:

a3 = Expand[a2^10];

Несмотря на то что вывод заблокирован, символу *a*3 значение все же присвоено:

Coefficient[a3,x^5y^25]
142506

Вычисленное выражение **CoefficientList[poly,form]** дает список коэффициентов при степенях *form* в полиноме *poly*, начиная с нулевой степени:

CoefficientList[Expand[a2^2],x]
$\{y^2 - 2y^3 - y^4 + 2y^5 + y^6,\ 2y - 4y^2 - 6y^3 + 8y^4 + 6y^5,$
$1 - 2y - 11y^2 + 12y^3 + 15y^4,\ -8y + 8y^2 + 20y^3,$
$-2 + 2y + 15y^2,\ 6y, 1\}$

Функция **Collect** в выражении **Collect[poly, x]** группирует члены с одной и той же степенью символа x:

Collect[Expand[(1 + x + y + z)^2],x]
$1 + x^2 + 2y + y^2 + 2z + 2yz + z^2 + x(2 + 2y + 2z)$

Эта же функция в выражении **Collect[poly,{x1,x2,...}]** группирует члены с одними и теми же степенями x_1, x_2, ...:

Collect[Expand[(1 + x + y + z)^2], {x, z}]

$1 + x^2 + 2y + y^2 + (2 + 2y)z + z^2 + x(2 + 2y + 2z)$

Если возникает необходимость узнать общее число слагаемых в многочлене *poly*, то это можно сделать, вычислив выражение **Length**[**poly**]. Если есть сомнение в том, что какое-то выражение *expr* есть многочлен по некоторой переменной, то это можно проверить с помощью **PolynomialQ**[**expr**, **var**], получая True, если *expr* полином по переменной *var*, и False иначе. Для нескольких переменных эта проверка осуществляется в виде **PolynomialQ**[**expr**, {**var**$_1$, **var**$_2$, ...}]. Выражение **PolynomialQ**[**expr**] проверяет, является ли *expr* полиномом относительно каких-либо переменных над полем рациональных чисел. Результат может быть False, например, в случае, когда *expr* содержит числа типа Real.

Функция **Variables**, примененная к *poly*, дает список всех независимых переменных в полиноме *poly*.

Важнейшими операциями при работе с полиномами являются нахождение наибольшего общего делителя и наименьшего общего кратного. Для полиномов $poly_1$ и $poly_2$ первое получается с помощью **PolynomialGCD**[**poly**$_1$, **poly**$_2$]. При вычислении этого выражения все символьные параметры в полиномах трактуются как переменные, и деление на них не допускается.

Вычисление выражения **PolynomialLCM**[**poly**$_1$, **poly**$_2$] дает наименьшее общее кратное, в то время как вычисление **PolynomialQuotient**[**poly**$_1$, **poly**$_2$] дает частное от деления $poly_1$ на $poly_2$, а вычисление **PolynomialRemainder**[**poly**$_1$, **poly**$_2$] — остаток от деления.

С помощью **Resultant**[**poly**$_1$, **poly**$_2$, **var**] можно найти результант полиномов по отношению к переменной *var*, т.е. произведение всех разностей $p_i - q_j$ корней этих полиномов (в случае, когда коэффициенты при старших степенях полиномов равны единице).

2.2. Подстановки

Одним из наиболее распространенных видов алгебраических преобразований являются подстановки, в результате выполнения которых какая-либо часть алгебраического выражения заменяется на новое выражение. В „Математике" существуют два способа делать подстановки. Один способ реализуется с помощью функции **Set**.

Вычисление выражения **Set[lhs,rhs]**, или **lhs = rhs** выполняется следующим образом. Сначала вычисляется выражение *rhs*, а затем вычисленное выражение присваивается как значение невычисленному выражению *lhs*. Всюду в дальнейшем *lhs*, в какие бы другие выражения оно ни входило, будет заменяться на вычисленное выражение *rhs*. Выражения *lhs* и *rhs* могут быть списками, и поэтому **Set** может быть задано в виде **{lhs₁,lhs₂,...} = {rhs₁,rhs₂,...}**. В этом случае вычисленные rhs_i будут присвоены lhs_i. В качестве примера присвоим символу x значение a и после вычислим выражение **a1**:

x = a
Expand[a1]
$4a^2$

Отменить глобальную подстановку в случае, когда выражение *lhs* есть символ, можно с помощью функции **Clear**: а именно **Clear[smbl₁,smbl₂,...]** отменяет все подстановки, связанные с символами $smbl_i$. Для одного символа можно использовать входную форму *smbl* = . выражения **Unset[smbl]**:

x = .
a1
$(-1+y)(x+y)+(x+y)^3$

Функция **Set** задает „глобальную" подстановку, оказывающую влияние на все последующие вычисления с ее первым аргументом. Если же подстановку одного выражения вместо другого

нужно сделать в одном конкретном выражении или же в нескольких, то по мере возникновения в этом необходимости применяется другой механизм, использующий функцию **Rule**.

Выражение **Rule[lhs,rhs]**, или **lhs → rhs**, задает правило, в соответствии с которым может быть сделана подстановка вычисленного выражения *rhs* вместо *lhs*. Выражение *rhs* вычисляется в момент задания **Rule**. Если *rhs* целесообразно вычислять в момент применения правила, то употребляется функция **RuleDelayed**, или **lhs :> rhs**.

После того как подстановка определена в виде правила **rl = x → a**, она может быть применена к конкретному выражению с помощью функций **ReplaceAll** (постфиксная форма /.):

ReplaceAll[Expand[(x + y + a)^2], rl], или
Expand[(x + y + a)^2] /. rl
$4a^2 + 4ay + y^2$

Подстановку целесообразно отдельно задавать и присваивать в качестве значения какому-либо символу только тогда, когда есть основания полагать, что она будет применяться несколько раз в ходе вычислений. Если же подстановка применяется один раз, то ее можно задать в виде второго аргумента функции **ReplaceAll** непосредственно в момент совершения подстановки:

(x + y + a)^2 /. x → a
$(2a + b)^2$

Функция **ReplaceAll** позволяет осуществить несколько подстановок одновременно. Тогда эти подстановки должны быть оформлены в виде списка и заданы вторым аргументом этой функции:

(x + y + a)^2 /. {x → a, y → b}
$(2a + b)^2$

Иногда возникает необходимость делать подстановки повторно. Предположим, что в выражении **Expand[(x + y + a)^2]** нужно избавиться от x и y, заменив x на a, y на ax, т.е. в конечном счете y на a^2. Тогда можно сделать либо подстановку $\{y \to ax, x \to a\}$, либо применить два раза подстановку $\{x \to a, y \to ax\}$. Последнее делают с помощью функции **ReplaceRepeated**, или в постфиксной форме $//.\{x \to a, y \to ax\}$:

Expand[(x + y + a)^2//.{x → a, y → ax}]

$4a^2 + 4a^3 + a^4$

После подстановки могут возникнуть выражения вида $(x^k)^n$, которые можно привести к виду x^{kn} с помощью функции **PowerExpand**. Эта функция преобразует (xk)^n в x^nk^n и $(x$^$k)$^n в x^(kn), какого бы вида ни было n. Преобразования, сделанные с помощью **PowerExpand**, корректны в общем случае, только если n есть целое, а x и k положительные.

2.3. Преобразования рациональных выражений

Дробь, числитель и знаменатель которой — полиномы, называется *рациональным выражением.* Уже знакомые нам функции **Expand** и **Factor** могут применяться к рациональным выражениям. Функция **Expand** раскрывает произведения и целые положительные степени в числителе и представляет рациональное выражение в виде суммы дробей, знаменатели которых совпадают со знаменателем исходного выражения, а числители есть отдельные слагаемые в раскрытом числителе:

ratnl = (x + y)(x^2 − y^2)/(x^3 − y^3)

$$\frac{(x+y)(x^2-y^2)}{x^3-y^3}$$

eratnl = Expand[ratnl]

$$\frac{x^3}{x^3-y^3}+\frac{x^2y}{x^3-y^3}-\frac{xy^2}{x^3-y^3}-\frac{y^3}{x^3-y^3}$$

Функция **Apart** раскладывает рациональное выражение на дроби с простыми знаменателями:

aratnl = Apart[ratnl]

$$1+\frac{xy}{x^2+xy+y^2}$$

Функция **Factor**, примененная к сумме рациональных выражений, приводит их к общему знаменателю и раскладывает на множители числитель и знаменатель полученного рационального выражения:

fratnl = Factor[aratnl]

$$\frac{(x+y)^2}{x^2+xy+y^2}$$

Функция **Together** приводит сумму рациональных выражений к общему знаменателю и сокращает общие множители в числителе и знаменателе:

tratnl = Together[aratnl]

$$\frac{x^2+2xy+y^2}{x^2+xy+y^2}$$

Общие множители в числителе и знаменателе рационального выражения не сокращаются автоматически. Чтобы их сократить, прибегают к функции **Cancel**:

Cancel[ratnl]

$$\frac{(x+y)^2}{x^2+xy+y^2}$$

Чтобы рассмотренная выше функция **Expand** действовала только на числитель или только на знаменатель рационального выражения, применяют модификации этой функции: **ExpandNumerator** и **ExpandDenominator**. Получить числитель и знаменатель рационального выражения можно с помощью функций **Numerator** и **Denominator**. Обе последние функции можно использовать для извлечения числителя и знаменателя рационального числа.

Для упрощения различных выражений, в том числе и рациональных, полезно применять функцию **Simplify**, которая выполняет последовательность алгебраических преобразований над выражением и приводит его к простейшей с точки зрения „Математики" форме. Рассматриваемую функцию удобно применять в постфиксной входной форме, т.е. приписывая ее с помощью // в конце входного выражения:

Together[aratnl] // Simplify

$$\frac{(x+y)^2}{x^2+x\,y+y^2}$$

2.4. Предикаты и булевы операции

Помимо того, что эти объекты важны сами по себе, они потребуются при обсуждении вопроса о способах задания и решения алгебраических и трансцендентных уравнений.

В „Математике" имеется семейство функций, принимающих булевы значения True и False. Заголовки этих функций оканчиваются на латинскую букву Q, хотя не все функции, заголовки которых оканчиваются на букву Q, булевозначны. Мы уже встречались с функциями **AtomQ** и **PolynomialQ**. Одними из наиболее употребительных булевозначных функций являются функции **IntegerQ, OddQ, EvenQ, PrimeQ**, которые проверяют свойства своих аргументов быть целыми, нечетными, четными или простыми числами, а **NumberQ** проверяет

свойство выражения быть числом любого типа. Рассмотренные функции называются *одноместными предикатами*, их вычисление всегда приводит к одному из двух булевых значений:

{OddQ[2], OddQ[5], OddQ[x^2+y^2]}
{False, True, False}

Имеется также класс функций одного аргумента, значения которых для некоторых аргументов могут быть True или False, а для других аргументов после вычисления их значение не является булевым. Например, функция **Positive** проверяет свойство числа быть положительным. Если число положительно, функция принимает значение True. Если не положительно — значение False. Если же после вычисления аргумента функции **Positive**, вычисленный аргумент не является числом, результат вычисления будет иметь заголовок Positive. Иными словами, булевозначные функции, оканчивающиеся на Q, определены для всех значений аргумента, а не оканчивающиеся на Q соответствуют так называемым частичным функциям, область определения которых не охватывает всего класса выражений „Математики“.

Двуместные предикаты сравнивают два выражения. Например, предикат **Less** сравнивает выражения по численной величине:

6 < 3
False

Подобно одноместным булевозначным функциям, двуместные предикаты могут принять значения, отличные от True или False:

x < y
$x < y$

Встроенными двуместными предикатами являются: **Equal** (==), **Unequal** (!=), **Less** (<), **Greater** (>), **GreaterEqual** (>=), **LessEqual** (<=). Предикат **Equal** принимает значение

True, только если внутренние формы вычисленных аргументов совпадают:

Sqrt[2] + 1/Sqrt[2] == 3/Sqrt[2]

$$\frac{1}{Sqrt[2]} + Sqrt[2] == \frac{3}{Sqrt[2]}$$

Мы видим: несмотря на то, что численно выражения слева и справа равны, предикат **Equal** не принимает значения True. Приведем выражение в правой части рассматриваемого равенства к общему знаменателю:

Together[Sqrt[2] + 1/Sqrt[2]] == 3/Sqrt[2]
True

Предикат **SameQ** (===), в отличие от **Equal**, всегда принимает либо значение True, либо False. А именно: если внутренние формы вычисленных значений аргументов совпадают, предикат принимает значение True, если нет — значение False.

Булева операция конъюнкции **And**, или логическое „И“, есть функция от нескольких (неопределенного числа) аргументов, последовательно вычисляющая свои аргументы и принимающая значение False как только очередной аргумент вычисляется на False. В остальных случаях значение этой функции не является булевым. Входная форма выражения **And**[e_1,e_2,...] имеет вид e_1 && e_2 && ... Булева операция дизъюнкции **Or**, логическое „Или“, входная форма e_1 || e_2 || ..., последовательно вычисляет свои аргументы и принимает значение True как только очередной аргумент при вычислении принимает значение True. В противном случае принимает небулево значение. Функция отрицания **Not**, логическое „Не“, определяется следующим образом: **Not[e]** принимает значение True, только если после вычисления выражение *e* принимает значение False, **Not[e]** принимает значение False, только если вычисленное *e* принимает значение True; и не определена как булевозначная операция в иных случаях.

2.5. Алгебраические и трансцендентные уравнения

Уравнения и методы получения их решений — один из важнейших разделов математики. Ниже мы рассмотрим основные функции, применяемые для нахождения символьных решений алгебраических и трансцендентных уравнений. Начнем со случая одного уравнения относительно одной неизвестной:

sols = Solve[x^3 - 2x + 1 == 0, x]

$$\left\{\{x \to 1\},\ \{x \to \frac{-1 - Sqrt[5]}{2}\},\ \{x \to \frac{-1 + Sqrt[5]}{2}\}\right\}$$

Уравнение $e = x^3 - 2x + 1 == 0$, записанное с помощью предиката **Equal**, является первым аргументом функции **Solve** — основной функции, применяемой для решения уравнений. Второй аргумент — переменная, относительно которой решается уравнение. Результатом вычисления выражения с заголовком **Solve** является список, элементами которого являются одноэлементные списки. Эти элементы есть подстановки вида $x \to expr_i$. Выражение $expr_i$ — это то, что обычно понимается под корнем уравнения. Если в уравнении e сделать подстановки, полученные в результате вычислений, то результатами будут True, если корни найдены правильно:

x^3 - 2x + 1 /. sols

$\{True, True, True\}$

Чтобы получить список корней, достаточно сделать подстановку:

x /. sols

$$\left\{1,\ \frac{-1 - Sqrt[5]}{2},\ \frac{-1 + Sqrt[5]}{2}\right\}$$

У рассмотренного уравнения достаточно простые корни, поэтому обратимся к новому примеру:

newsols = Solve[x^3 − 3x + 1 == 0, x]

$$\left\{\left\{x \to \frac{3\ 2^{\frac{1}{3}}}{(-27+27I\,Sqrt[3])^{\frac{1}{3}}} + \frac{(-27+27I\,Sqrt[3])^{\frac{1}{3}}}{3\ 2^{\frac{1}{3}}}\right\},\right.$$

$$\left\{x \to \frac{-3(1+I\,Sqrt[3])}{2^{\frac{2}{3}}(-27+27I\,Sqrt[3])^{\frac{1}{3}}} - \frac{(1-I\,Sqrt[3])(-27+27I\,Sqrt[3])^{\frac{1}{3}}}{6\,2^{\frac{1}{3}}}\right\},$$

$$\left.\left\{x \to \frac{-3(1-I\,Sqrt[3])}{2^{\frac{2}{3}}(-27+27I\,Sqrt[3])^{\frac{1}{3}}} - \frac{(1+I\,Sqrt[3])(-27+27I\,Sqrt[3])^{\frac{1}{3}}}{6\,2^{\frac{1}{3}}}\right\}\right\}$$

В полученных формулах I — встроенная константа, мнимая единица i. На первый взгляд, все три корня — комплексные числа с не равными нулю мнимыми частями. На самом деле корни вещественные. Чтобы в этом убедиться, воспользуемся функцией **N**, дающей приближенное численное значение выражений, и функцией **Plot**, рисующей графики. Вычислим выражение:

N[newsols]

$\{\{x \to 1.53209 - 3.33067\ 10^{-16}I\},$

$\{x \to -1.87939 - 3.88578\ 10^{-16}I\},$

$\{x \to 0.347296 + 9.99201\ 10^{-16}I\}\}$

Результат показывает, что мнимые части корней имеют одинаковый порядок 10^{-16}. Попробуем увеличить точность вычислений. Для этого воспользуемся тем, что второй необязательный аргумент у функции **N** задает число значащих цифр вещественных чисел.

N[newsols, 25]

$\{\{x \to 1.532088886237956070404785 + 0.10^{-32}I\},$

$\{x \to -1.879385241571816768108219 + 0.10^{-32} I\}$,

$\{x \to 0.347296355333860697703433 - 0.10^{-32} I\}\}$

Теперь порядок мнимых частей корней 10^{-32}. Это говорит о том, что корни — вещественные. Наглядное представление о расположении корней можно получить, нарисовав график функции $x^3 - 3x + 1$ с помощью функции **Plot** (рис. 2.1):

Plot[x^3 - 3x + 1, {x, -2, 2}]

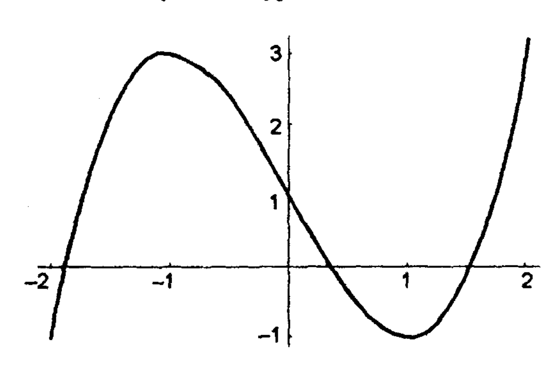


Рис. 2.1

График наглядно показывает, что кубическая парабола $x^3 - 3x + 1$ пересекает ось абсцисс в трех точках. Присутствие в **newsols** мнимой единицы есть результат следующего выбора ветви многозначной функции $z^{(1/k)}$ комплексного аргумента z. Аргумент φ любого комплексного числа считается изменяющимся от $-\pi$ до π, поэтому $z^{(1/k)}$ имеет аргумент φ/k. Например, $(-1)^{(1/3)}$, т.е. кубический корень из минус единицы, равен $1/2 + I\sqrt{3}/2$.

Иное по сравнению с **Solve** представление для корней уравнений дает функция **Roots**.

rts = Roots[e, x]

$$x = 1 \;||\; x = \frac{-1 - Sqrt[5]}{2} \;||\; x = \frac{-1 + Sqrt[5]}{2}$$

Второе представление можно легко преобразовать в первое с помощью функции **ToRules**:

ToRules[rts]

$$\left\{\{x \to 1\}, \{x \to \frac{-1 - Sqrt[5]}{2}\}, \{x \to \frac{-1 + Sqrt[5]}{2}\}\right\}$$

Если функция **Solve** не может найти корни уравнения в явном виде, то результат вычисления корней выглядит следующим образом:

Solve[x^5 – 3x + 1 == 0, x]

$\{ToRules[Roots[-3x + x^5 == -1, x]]\}$

Системы уравнений задаются в виде списка **syslist** = {**x^2 + + y^2 == a^2, y – x == 1**} либо уравнения соединяются с помощью конъюнкции **sysand** = {**x^2 + y^2 == a^2** && **y – – x == 1**}. Результат в обоих случаях один и тот же:

Solve[syslist, {x,y}] или **Solve[sysand, {x,y}]**

$$\left\{\{x \to \frac{-1 - Sqrt[-1 + 2a^2]}{2}, y \to \frac{2 - Sqrt[4 - 8(1 - a^2)]}{4}\},\right.$$
$$\left.\{x \to \frac{-1 + Sqrt[-1 + 2a^2]}{2}, y \to \frac{2 + Sqrt[4 - 8(1 - a^2)]}{4}\}\right\}$$

Иногда возникает необходимость исключить часть переменных из системы уравнений и найти уравнения для оставшихся переменных. Для этого пользуются функцией **Eliminate**:

elmsys = Eliminate[syslist, y]

$a^2 == 1 + 2x + 2x^2$

Полученное уравнение можно решить относительно переменной x:

Solve[elmsys, x]

$$\left\{\left\{x \to \frac{-2 - Sqrt[4 - 8(1 - a^2)]}{4}\right\},\right.$$
$$\left.\left\{x \to \frac{-2 + Sqrt[4 - 8(1 - a^2)]}{4}\right\}\right\}$$

Тот же результат можно получить, указав в функции **Solve** третий аргумент — исключаемую переменную (или список исключаемых переменных):

Solve[syslist, x, y]

$$\left\{\left\{x \to \frac{-2 - Sqrt[4 - 8(1 - a^2)]}{4}\right\},\right.$$
$$\left.\left\{x \to \frac{-2 + Sqrt[4 - 8(1 - a^2)]}{4}\right\}\right\}$$

С помощью функции **Solve** могут быть найдены решения некоторого, впрочем, весьма ограниченного класса трансцендентных уравнений или систем уравнений. При вычислении корней таких уравнений „Математика" печатает на экране предупреждение о том, что в процессе решения она обращается к обратным функциям (и ей приходится выбирать интервалы монотонности встречающихся в уравнении функций), поэтому не все решения могут быть найдены:

Solve[Sin[x]^2 – Sin[x] + a == 0, x]

Solve:: ifun: Warning: Inverse functions are being used by Solve, so some solutions may not be found.

$$\left\{\left\{x \to ArcSin\left(\frac{1 - Sqrt[1 - 4a]}{2}\right)\right\},\right.$$
$$\left.\left\{x \to ArcSin\left(\frac{1 + Sqrt[1 - 4a]}{2}\right)\right\}\right\}$$

Функция **Solve** находит решения уравнений, трактуя их „в общем положении", т.е. для неспецифических значений входящих в уравнения символьных параметров:

```
Solve[a x^2 + b x + c == 0, x]
```

$$\left\{\left\{x \to \frac{-b - Sqrt[b^2 - 4ac]}{2a}\right\}, \left\{x \to \frac{-b + Sqrt[b^2 - 4ac]}{2a}\right\}\right\}$$

Очевидно, что при $a = 0$ эти выражения для корней не имеют смысла. Функция **Reduce** позволяет учесть подобные особые значения параметров. Вычисляя выражение **Reduce**[*eqns*, *vars*], „Математика“ упрощает уравнения *eqns* и порождает уравнения, эквивалентные *eqns* и содержащие все возможные решения:

```
Reduce[a x^2 + b x + c == 0, x]
```

$$a \neq 0 \ \&\& \ x = \frac{-b - Sqrt[b^2 - 4ac]}{2a} \ ||$$

$$a \neq 0 \ \&\& \ x = \frac{-b + Sqrt[b^2 - 4ac]}{2a} \ ||$$

$$c = 0 \ \&\& \ b = 0 \ \&\& \ a = 0 \ || \ b \neq 0 \ \&\& \ x = -\frac{c}{b} \ \&\& \ a = 0$$

В прикладной математике очень важную роль играют системы линейных алгебраических уравнений. Разумеется, их решения можно находить с помощью функции **Solve**. Однако часто линейные системы возникают в процессе решения каких-то прикладных задач, в которых генерируются не сами системы, а матрицы их коэффициентов. В подобных случаях удобнее воспользоваться функцией **LinearSolve**. Предварительно отметим, что матрицы в „Математике“ задаются с помощью списков построчно:

```
m = {{a1, a2}, {b1, b2}}
```

$\{\{a1, a2\}, \{b1, b2\}\}$

Стандартное представление матрицы может быть получено с помощью функции **MatrixForm**.

```
m // MatrixForm
```

$$\begin{matrix} a1 & a2 \\ b1 & b2 \end{matrix}$$

Функция **LinearSolve** имеет два аргумента. Первый — матрица коэффициентов системы, а второй — список ее правых частей.

LinearSolve[m, c]

$$\left\{\frac{-(b2\ c1)+a2\ c2}{a2\ b1-a1\ b2}, \frac{-(b1\ c1)+a1\ c2}{-(a2\ b1)+a1\ b2}\right\}$$

Здесь $c = \{c1, c2\}$ — вектор правых частей системы. Функция **LinearSolve** наиболее эффективна в случае систем большой размерности с разреженными матрицами коэффициентов.

2.6. Математический анализ

Одну из фундаментальных операций математического анализа — дифференцирование — осуществляют две функции: **D** и **Dt**. Выражение **D[expr, x]** имеет результатом (частную) производную выражения **expr** по переменной x. При вычислении смешанных и кратных производных второй и т.д. аргументы функции **D** могут быть заданы в виде списков вида $\{var_i, k_i\}$, где k_i — порядок производной по переменной var_i:

dif = D[x^3 Sin[x^2] E^(-x), {x, 2}]

$$\frac{14x^3 Cos[x^2]}{E^x} - \frac{4x^4 Cos[x^2]}{E^x} + \frac{6x\, Sin[x^2]}{E^x} - \frac{6x^2 Sin[x^2]}{E^x} + \frac{x^3 Sin[x^2]}{E^x} - \frac{4x^5 Sin[x^2]}{E^x}$$

Вычислена вторая производная от функции $x^3 \sin x^2 e^{-x}$, заданной в виде явной аналитической формулы. Рассмотрим произвольную функцию $f[x, y]$ и вычислим ее третью смешанную производную $\partial^3 f / \partial x \partial y^2$:

D[f[x, y], x, {y, 2}]

$f^{(1,2)}[x, y]$

Ответ записан с помощью мультииндекса (1,2), компоненты которого показывают, что вычислена первая производная по

первому аргументу и вторая производная по второму аргументу. Модуль мультииндекса, равный сумме его компонент, указывает порядок производной. Внутренняя форма вычисленного выражения **dif2** есть

dif2 // FullForm
Derivative[1,2][*f*][*x*, *y*]

Здесь нам встретилось выражение, заголовок которого не является символом. „Математика" всегда сводит производные, которые она не смогла явно вычислить, к выражениям с заголовком Derivative[$n_1, n_2, \ldots$][f].

Любое выражение, не содержащее символов, по которым производится дифференцирование, функция **D** рассматривает как константу по переменным дифференцирования:

D[a Sin[x], x]
a Cos[*x*]

В отличие от функции **D** функция **Dt**, *полная* производная, рассматривает все символы, входящие в дифференцируемое выражение как функции от переменных дифференцирования:

Dt[a Sin[x], x]
a Cos[*x*] + *Dt*[*a*, *x*] *Sin*[*x*]

Если второй аргумент у функции **Dt** не указывается, то **Dt[expr]** понимается как дифференциал выражения *expr*:

Dt[a Log[y]]

$$\frac{a\,Dt[y]}{y} + Dt[a]\,Log[y]$$

Операцией, обратной дифференцированию, является нахождение первообразной. Соответствующая функция „Математики" — **Integrate**:

Integrate[dif, x]

$$\frac{2x^4 Cos[x^2]}{E^x} + \frac{3x^2 Sin[x^2]}{E^x} - \frac{x^3 Sin[x^2]}{E^x}$$

Заметим, что константа интегрирования отсутствует. Это означает, что в общем случае **Integrate** вычисляет локальную первообразную, т.е. функцию, которая может иметь скачки в дискретном множестве точек (см. упр. 4 к этой главе).

Для вычисления определенного интеграла в качестве второго аргумента функции **Integrate** указывается список $\{x, a, b\}$, первым элементом которого является переменная интегрирования, вторым элементом — нижний, третьим элементом — верхний предел интегрирования:

Integrate[dif, {x, 0, Sqrt[Pi]]

$$\frac{-2Pi^2}{E^{Sqrt[Pi]}}$$

Здесь Pi есть встроенная константа — число π. Функция **Integrate** позволяет вычислять повторные интегралы. В последних пределы интегрирования по переменным даются в том порядке, какой принят в математическом анализе:

Integrate[x + y, {x, 0, 2a}, {y, 0, x}]

$4a^3$

В примере вычислен кратный интеграл от функции $x + y$ по треугольной области $\Omega = \{(x, y) \,|\, 0 \leq x \leq 2a, 0 \leq y \leq x\}$.

Разложение функций в ряды Тейлора осуществляет функция **Series**. При вычислении выражения **Series[f, {x, x0, n}]** находится сумма первых $n + 1$ членов разложения функции f в ряд Тейлора в окрестности точки x_0:

{Series[f[x], {x, 0, 3}], Series[E^(-x^2)Log[1 + x], {x, 0, 3}]}

$$\{f[0] + f'[0]x + \frac{f''[0]x^2}{2} + \frac{f^{(3)}[0]x^3}{6} + O[x]^4,$$

$$x - \frac{x^2}{2} - \frac{2x^3}{3} + O[x]^4\}$$

Для функций от нескольких переменных функция **Series** позволяет находить последовательные разложения в ряды Тейлора по различным независимым переменным:

Series[f[x,y],{x,a,1},{y,b,1}]

$$f[a,b]+f^{(0,1)}[a,b](-b+y)+O[-b+y]^2+(f^{(1,0)}[a,b]+ + f^{(1,1)}[a,b](-b+y)+O[-b+y]^2)(-a+x)+O[-a+x]^2$$

В качестве точки, в окрестности которой производится разложение в ряд Тейлора, вполне может быть выбрана бесконечно удаленная точка **Infinity**, кроме того, функция **Series** учитывает наличие особенностей у функции:

{Series[E^(1/x),{x,Infinity,2}],
Series[Cos[x]Log[x],{x,0,2}]}

$$\left\{1+\frac{1}{x}+\frac{1}{2x^2}+O[\frac{1}{x}]^3,\ Log[x]-\frac{Log[x]x^2}{2}+O[x]^3\right\}$$

С разложениями в ряд Тейлора можно проделывать обычные алгебраические операции: перемножать, возводить в степень, вычислять функции от разложений, дифференцировать, интегрировать, брать композицию рядов и т.д., оставаясь в том же классе выражений:

{s1 = Series[E^Sin[x],{x,0,2}],
s2 = Series[ArcSin[x],{x,0,3}]}

$$\{1+x+\frac{x^2}{2}+O[x]^3,\ x+\frac{x^3}{6}+O[x]^4\}$$

{s1 s2, D[s1,x], Integrate[s2,x], s1 /. x → s2}

$$\left\{x+x^2+\frac{2x^3}{3}+O[x]^4,\ 1+x+O[x]^2,\ \frac{x^2}{2}+\frac{x^4}{24}+O[x]^5,\right.$$
$$\left.1+x+\frac{x^2}{2}+O[x]^3\right\}$$

С помощью функции **InverseSeries** можно обращать ряды:

{s3 = InverseSeries[s2, x], s2 /. x → s3}

$x - \frac{x^3}{6} + O[x]^4,\ x + O[x]^4$

Функция **Series** имеет результатом выражение „Математики", имеющее заголовок SeriesData, поэтому функции, используемые для преобразования алгебраических выражений, не действуют на этот результат. Вычисленное выражение SeriesData$[x, x_0, \{a_0, a_1, \ldots\}, nmin, nmax, den]$ представляет отрезок степенного ряда по переменной x в окрестности точки x_0. Величины a_i есть коэффициенты ряда. Разность $x - x_0$ входит в ряд в степенях $nmin/den$, $(nmin + 1)/den$, …, $nmax/den$:

ser1 = SeriesData[x, 0, {1, 2, 3}, 1, 4, 3]

$x^{\frac{1}{3}} + 2x^{\frac{2}{3}} + 3x + O[x]^{\frac{4}{3}}$

Для получения списков коэффициентов ряда часто используют функцию **Table**. Например, список {1, 2, 3} является значением выражения **Table[i, {i, 3}]**. Если нужно получить ряд с неопределенными символьными коэффициентами, можно вычислить выражение **Table[a[i], {i, 3}]** или выражение **Array[a, 3]**:

{Table[i, {i, 3}], Table[a[i], {i, 3}], Array[a, 3]}

$\{\{1, 2, 3\},\ \{a[1], a[2], a[3]\},\ \{a[1], a[2], a[3]\}\}$

Воспользуемся функцией **Array**, чтобы получить ряд ser1, в котором коэффициенты 1,2,3 заменены на $a[1]$, $a[2]$, $a[3]$:

ser2 = SeriesData[x, 0, Array[a, 3], 1, 4, 3]

$a[1]x^{\frac{1}{3}} + a[2]x^{\frac{2}{3}} + a[3]x + O[x]^{\frac{4}{3}}$

Функция **Normal** конвертирует ряды в многочлены, одновременно отбрасывая символ $O[x]^n$.

2.7. Специализированные программы

Описывая различные функции „Математики", мы употребляли определение „встроенные". Оно означает, что эти функции становятся непосредственно доступными после загрузки „Математики". Встроенных функций обычно оказывается вполне достаточно для выполнения широкого круга вычислений. Однако при работе в какой-то специализированной области могут понадобится иные функции, определенные в отдельно хранящихся специализированных программах и существенно расширяющие возможности пользователя. Такие программы сосредоточены в 13 пакетах, или поддиректориях, директории PACKAGES. Вот их названия:

Algebra	Miscellaneous
Calculus	NumberTheory
DiscreteMath	NumericalMath
Examples	ProgrammingExamples
Geometry	Statistics
Graphics	Utilities
LinearAlgebra	

Все эти пакеты также содержат поддиректории. Например, пакет Graphics содержит поддиректории FilledPlot, ImplicitPlot, Polyhedra и т.д. Пакет Calculus содержит поддиректории DSolve, FourierTransform, а также LaplaceTransform, VariationalMethods, VectorAnalysis и т.д. Если требуется вычислить преобразование Фурье каких-либо функций, то следует подгрузить поддиректорию FourierTransform, находящуюся в пакете Calculus. Для этой цели можно воспользоваться функциями **Needs** или **Get**.

Needs["Calculus‘FourierTransform‘"]

или

Get["Calculus‘FourierTransform‘"]

Последнее выражение также можно ввести в виде

<< Calculus‘FourierTransform‘

После подгрузки пакета становятся доступными функции, которые в нем определены и имена которых можно получить, используя функцию **Names**. Пример:

Names["Calculus‘FourierTransform‘*"]

Теперь мы имеем возможность вычислять преобразование Фурье функций:

FourierTransform[E^ - x^2 Sin[a x], x, p]

$$\frac{\frac{I}{2}Sqrt[Pi]}{E^{\frac{(a-p)^2}{4}}} - \frac{\frac{I}{2}Sqrt[Pi]}{E^{\frac{(a+p)^2}{4}}}$$

При решении систем алгебраических уравнений с двумя неизвестными бывает полезна функция **ImplicitPlot**, определенная в программе ImplicitPlot директории Graphics. В отличие от функции **Plot** она позволяет получать графики неявных функций $y = f_i(x)$, где $f_i(x)$ находятся из отдельных уравнений системы (рис. 2.2).

ImplicitPlot[{x^3 + y^2 == 1, y^2 - x == 0}, {x, -1, 2}]

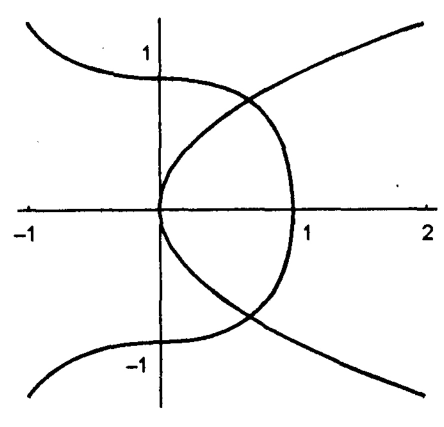


Рис. 2.2

Если в процессе вычислений требуются функции, определенные в различных поддиректориях одной и той же директории, то удобно воспользоваться следующей формой подгрузки функций. Следует обратиться к программе Master, имеющейся в каждой директории, выполнив команду

<< Directory'Master',

где Directory есть имя нужной директории. Если происходит обращение к какой-то функции директории, то программа Master автоматически загружает ту поддиректорию, в которой используемая функция фактически определена.

2.8. Обыкновенные дифференциальные уравнения

Основная функция, используемая для нахождения решений обыкновенных дифференциальных уравнений в символьной форме, есть **DSolve**:

$$\mathbf{sol1 = DSolve[y''[x] + x\,y'[x] == 0, y[x], x]}$$

$$\{\{y[x] \to C[1] + Sqrt[\frac{Pi}{2}]C[2]\,Erf[\frac{x}{Sqrt[2]}]\}\}$$

Первый аргумент функции **DSolve** — обыкновенное дифференциальное уравнение, в котором производные от искомой функции $y[x]$ записаны в специальной входной форме функции **Derivative**. Ответ содержит две произвольные постоянные C[1] и C[2]. Если попытаться проверить правильность полученного решения с помощью подстановки в исходное уравнение, то мы встретимся со следующей трудностью:

$$\mathbf{ode1 = y''[x] + x\,y'[x] == 0;\ ode1\ /.\ sol1}$$

$$\{x\,y'[x] + y''[x] == 0\}$$

Таким образом, производные от подстановки *sol*1 не вычисляются автоматически. Проверку правильности следует производить в виде

ode1 /. D[sol1,x,x] /. D[sol1,x]
{{*True*}}

Встроенная функция **DSolve** предназначена для решения относительно простых дифференциальных уравнений. Например, если попытаться с ее помощью решить уравнение $x^2y''[x] - y'[x]^2 = 0$, то получается следующий результат:

DSolve[x^2y''[x] - y'[x]^2 == 0,y[x],x]
DSolve:: dnim: Built-in procedures cannot solve this differential equation
$\mathrm{DSolve}[-y'[x]^2 + x^2y''[x] == 0, y[x], x]$

Гораздо более мощная функция с тем же названием **DSolve** определена в поддиректории DSolve директории Calculus. После ее подгрузки рассматриваемое дифференциальное уравнение легко разрешается:

DSolve[x^2y''[x] - y'[x]^2 == 0,y[x],x]

$$\left\{\{y[x] \to -(\frac{x}{C[1]}) + C[2] - \frac{Log[1 - xC[1]]}{C[1]^2}\}\right\}$$

Дифференциальное уравнение может быть снабжено дополнительными условиями в форме задачи Коши:

DSolve[{y''[x] + xy'[x] == 0,y[0] == 0,y'[0] == 1},y[x],x]

$$\left\{\{y[x] \to \frac{Sqrt[Pi]}{2} Erf[\frac{x}{Sqrt[2]}]\}\right\}$$

С помощью функций **DSolve**, как встроенной, так и подгружаемой, можно решать также и системы дифференциальных уравнений:

DSolve[{x'[t] == y[t], y'[t] == -a^2x[t],
x[0] == 1, y[0] == 0}, {x[t], y[t]}, t]

$$\left\{\{x[t] \to \frac{E^{-Iat}}{2} + \frac{E^{Iat}}{2},\ y[t] \to \frac{-I}{2} aE^{-Iat} + \frac{I}{2} aE^{Iat}\}\right\}$$

В примере мы воспользовались встроенной функцией. Если необходимо нарисовать графики полученных решений или фазовый портрет динамической системы, то второй аргумент у функции **DSolve** следует задать в следующем виде:

sol2 = DSolve[{x'[t] == y[t], y'[t] == -a^2x[t],
x[0] == 1, y[0] == 0}, {x, y}, {t}]

$$\left\{\{x \to Function[t, \frac{e^{-Iat}}{2} + \frac{E^{Iat}}{2}],\right.$$
$$\left. y \to Function[t, \frac{-I}{2} aE^{-Iat} + \frac{I}{2} aE^{Iat}]\}\right\}$$

Последний ответ представлен с помощью так называемых *чистых функций*, с которыми можно будет познакомиться позднее. Сейчас мы просто воспользуемся этим результатом для того, чтобы нарисовать графики функций $x[t]$ и $y[t]$ (рис. 2.3).

Plot[Evaluate[{x[t], y[t]} /. sol2 /. a → 1/2], {t, -2Pi, 2Pi}]

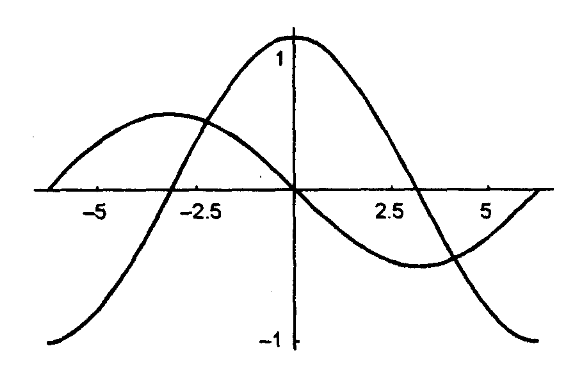


Рис. 2.3

Фазовый портрет можно нарисовать с помощью функции **ParametricPlot** (рис. 2.4):

ParametricPlot[Evaluate[{x[t],y[t]} /. sol2 /. a → 1/2], {t, −2Pi, 2Pi}, AspectRatio → Automatic]

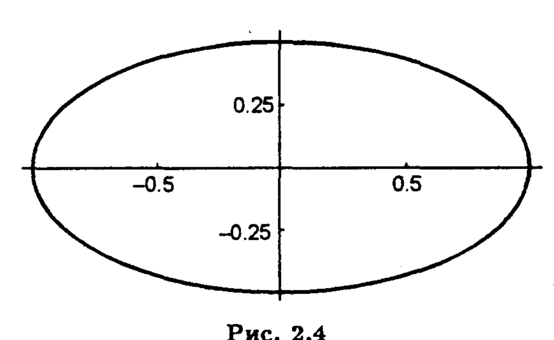


Рис. 2.4

Даже усиленная версия функции **DSolve** не может найти символьных решений всех обыкновенных дифференциальных уравнений. Встроенной функцией „Математики“, реализующей один из численных методов решения обыкновенных дифференциальных уравнений, является функция **NDSolve**. Проанализируем, используя **NDSolve**, нелинейные колебания системы, параметры которой изменяются со временем. Пусть система описывается уравнениями: $\dot{x}(t) = y(t)$, $\dot{y}(t) = -x(t) - 0.7\cos(0.3t)\sin x(t)$. Поставим начальные условия $x(0) = 1$, $y(0) = 0$ и вычислим выражение:

nlosc = NDSolve[{x′[t] == y[t],
y′[t] == −x[t] − 0.7Cos[0.3t]Sin[x[t]],
x[0] == 1, y[0] == 0}, {x, y}, {t, 0, 3Pi}]
{{$x \to$ *InterpolatingFunction*[{0, 9.42478}, <>],
$y \to$ *InterpolatingFunction*[{0, 9.42478}, <>]}}

Результатом вычисления являются две интерполяционные функции для $x(t)$ и $y(t)$ на интервале (0, 9.42478). С их

помощью можно вычислить значения решения в любой точке этого интервала:

```
{x[1],y[1]} /. nlosc
{{0.299012,-1.21722}}
```

или нарисовать фазовый портрет колебаний (рис. 2.5):

```
ParametricPlot[Evaluate[{x[t],y[t]} /. nlosc],{t,0,3Pi},
AspectRatio -> Automatic]
```

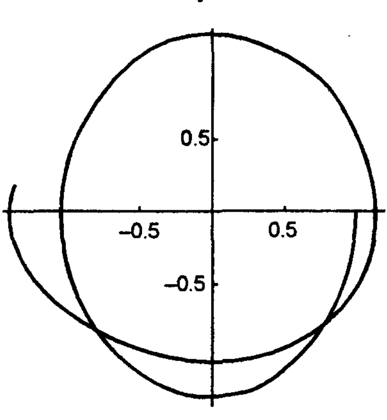


Рис. 2.5

Еще один способ получения приближенных решений, но уже в символьной форме — это метод рядов. Найдем приближенное ограниченное решение дифференциального уравнения $xy'(x) - y(x)^2 - x = 0$ в окрестности особой точки $x = 0$ в виде отрезка степенного ряда с точностью $O[x]^7$. Задача сводится к определению семи коэффициентов в формуле $y(x) = a_0 + a_1x + \ldots + a_6x^6 + O[x]^7$. Поскольку решение ищется в виде отрезка ряда, определим объект:

```
y[x_] := SeriesData[x,0,Array[a,7,0]]
```

Обратим внимание на два обстоятельства. Во-первых, мы использовали так называемое отложенное определение, поэтому

выходной строчки нет. Во-вторых, в определении использована одна очень важная конструкция „Математики“, называемая именованным шаблоном — $x_$. Эта конструкция будет подробно рассмотрена в дальнейшем. Посмотрим, как выглядит $y[x]$:

y[x]

$a[0]+a[1]x+a[2]x^2+a[3]x^3+a[4]x^4+a[5]x^5+a[6]x^6+O[x]^7$

Теперь вычислим дифференциальное уравнение с полученным выражением для $y[x]$:

ode2 = x y'[x] − y[x]^2 − x == 0

$-a[0]^2+(-1+a[1]-2a[0]a[1])x+(-a[1]^2+2a[2]-$
$-2a[0]a[2])x^2+(-2a[1]a[2]+3a[3]-2a[0]a[3])x^3+(-a[2]^2-$
$-2a[1]a[3]+4a[4]-2a[0]a[4])x^4+(-2a[2]a[3]-2a[1]a[4]+$
$+5a[5]-2a[0]a[5])x^5+(-a[3]^2-2a[2]a[4]-2a[1]a[5]+$
$+6a[6]-2a[0]a[6])x^6+O[x]^7==0$

Уравнения для коэффициентов $a[i]$ получаются приравниванием нулю коэффициентов при различных степенях переменной x. Это можно сделать, применив к *ode*2 функцию **LogicalExpand**:

eqcoefy = LogicalExpand[ode2]

$-a[0]^2==0$ && $-1+a[1]-2a[0]a(1)==0$ &&
$-a[1]^2+2a[2]-2a[0]a[2]==0$ &&
$-2a[1]a[2]+3a[3]-2a[0]a[3]==0$ &&
$-a[2]^2-2a[1]a[3]+4a[4]-2a[0]a[4]==0$ &&
$-2a[2]a[3]-2a[1]a[4]+5a[5]-2a[0]a[5]==0$ &&
$-a[3]^2-2a[2]a[4]-2a[1]a[5]+6a[6]-2a[0]a[6]==0$

Легко видеть, что рассматриваемые уравнения имеют единственное решение, которое можно найти с помощью знакомой нам функции **Solve**:

coefy = Solve[eqcoefy, Table[a[i], {i, 0, 6}]]

$$\left\{\left\{a[6] \to \frac{473}{4320},\ a[5] \to \frac{19}{120},\ a[4] \to \frac{11}{48},\ a[3] \to \frac{1}{3},\ a[2] \to \frac{1}{2},\ a[0] \to 0,\ a[1] \to 1\right\}\right\}$$

Теперь можно найти само приближенное решение:

y[x] /. coefy

$$\left\{x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{11x^4}{48} + \frac{19x^5}{120} + \frac{473x^6}{4320} + O[x]^7\right\}$$

2.9. Числа и операции над числами

Напомним, что в „Математике" четыре типа чисел: целые, рациональные, вещественные и комплексные. Все они могут содержать любое количество цифр. Арифметические операции над первыми двумя типами чисел выполняются абсолютно точно (все цифры результата верные), что является характерной особенностью компьютерных алгебр:

sum = 357784905111127390538463709587618 +
+ 115634553279325911771666205734605
473419458390453302310129915322223

В этом примере найдена сумма двух тридцатитрехзначных целых чисел. Результат возведения в пятидесятую степень числа пять, найденный „Математикой", равен:

$5^{50} = 88817841970012523233890533447265625.$

Дадим пример вычисления более сложного арифметического выражения:

ar = 8353 * 644 − 315^14 + 564/1025(314 − 276)

$$-\left(\frac{970670024879503146412062757818396788893}{1025}\right)$$

В форматах целых или рациональных чисел трудно получить представление о величине результата. Кроме того, они неудобны в численных расчетах еще и потому, что „Математика", стремясь сохранить тип чисел, оставляет фактически невычисленными такие выражения, как Sqrt[10], Sin[Pi/5] и т.д.

{Sqrt[20], Sin[Pi/5]}

$\left\{2\,Sqrt[5],\ Sin[\frac{Pi}{5}]\right\}$

Если в последнем примере заменить целое число 20 на вещественное число 20. (характерной особенностью формата вещественных чисел является присутствие десятичной точки в записи числа), то получится следующий результат:

Sqrt[20.]

4.47214

Функция **N** является конвертором в формат вещественных чисел или в формат комплексных чисел с вещественными компонентами, в которых результат записывается в более привычном виде, принятом в научных и инженерных расчетах:

{N[sum], N[ar]}

$\{4.7341910^{32},\ -9.4699510^{34}\}$

При вычислениях с вещественными числами очень полезны функции **Precision** и **Accuracy**, которые весьма условно можно перевести как *разрядность* и *точность*. Разрядность есть число десятичных цифр, используемых „Математикой" в представлении вещественных чисел при вычислениях, в то время как точность есть число десятичных цифр, используемых для представления дробной части вещественных чисел. С вычислительной точки зрения вещественные числа трактуются „Математикой" либо как имеющие машинную разрядность (по умолчанию), либо как имеющие неопределенную разрядность. Для того чтобы узнать машинную разрядность

для вашего компьютера, достаточно вычислить выражение *$MachinePrecision*. На PC-компьютере автора это число равно 16.

{Precision[1./7], **Accuracy**[1./7]}
{16, 17}

Значения рассматриваемых функций для целых и рациональных чисел равно *Infinity*. Если пользователь ввел вещественное число, использовав больше 16 цифр, то оно трактуется как имеющее разрядность, большую, чем разрядность введенного числа, с дополнительными неизвестными цифрами. При вычислениях с такими числами производится анализ, на какие разряды результата вычислений могут воздействовать дополнительные цифры введенного числа. Такие разряды не включаются в результат.

У функции **N** имеется второй, необязательно указываемый аргумент, который определяет разрядность чисел, используемую при вычислении результата. Если второй аргумент не задан, то, по умолчанию, в вычислениях используется *$MachinePrecision* цифр, а на экране компьютера результат представлен с шестью цифрами. К функции **N** обращаются для нахождения численных значений встроенных констант **E**,**Pi**, и др.:

pin = N[Pi,40]
3.1415926535897932384626433832795028841972

sqad = N[Sqrt[20],25]
4.472135954999579392818347

{**Precision**[**pin**], **Accuracy**[**pin**],
Precision[**sqad**], **Accuracy**[**sqad**]}
{40, 40, 25, 24}

Встроенные алгоритмы „Математики“ стремятся, чтобы при вычислении функции **N** от встроенных констант все цифры

результата были верные. Однако в общем случае не следует ожидать, что все n цифр результата вычисления выражения **N[expr, n]** верные. Справедливо лишь то, что вычисления велись с заданной пользователем разрядностью, равной n.

Аргументами функций **N**,**Precision** и **Accuracy** могут быть и комплексные числа:

compl = N[Sqrt[20 + 5I]]
4.50641 + 0.554765*I*

{Precision[compl], Accuracy[compl]}
{16, 15}

Если в процессе интерактивных вычислений возникают вещественные числа, величиной меньшие, чем 10^{-10}, то с помощью функции **Chop** их можно отбросить:

{exp = E^(IN[Pi]), Chop[exp]}
$\{-1. + 1.2251510^{-16},\ -1.\}$

Задание второго числового аргумента ε у функции **Chop** позволяет отбрасывать вещественные числа, по модулю меньшие ε.

Укажем несколько функций в известном смысле обратных по отношению к функции **N**, т.е. сопоставляющих близкие к вещественным числам целые или рациональные числа. Это прежде всего функции, вычисляющие целую часть числа a, ближайшее к a целое, и ближайшее целое, превосходящее a, — **Floor[a]**,**Round[a]**, **Ceiling[a]**:

{Floor[5.7], Round[5.7], Ceiling[5.7]}
{5, 6, 6}

Функция **Rationalize** конвертирует вещественные числа в рациональные. Точность преобразования может быть задана в качестве второго аргумента этой функции:

Rationalize[N[Pi], 0.001]

$$\frac{355}{113}$$

Интересно вычислить список нескольких, все более точных приближений числа **Pi** рациональными числами:

Table[Rationalize[N[Pi], (0.1)^i], {i, 7}]

$$\left\{\frac{22}{7}, \frac{22}{7}, \frac{355}{113}, \frac{355}{113}, \frac{355}{113}, \frac{355}{113}, \frac{104348}{33215}\right\}$$

Упражнения

1. *Реализуйте* с помощью „Математики" следующий способ решения уравнения четвертой степени $x^4 + 2x^3 - 6x^2 + 2x + 1 = 0$. Сначала обе части уравнения делятся на x^2, а затем вводится новое неизвестное y, связанное с x соотношением $y = x + 1/x$. Относительно y получается квадратное уравнение. Находятся его решения, а затем x. *Проверьте* правильность полученных таким способом корней исходного уравнения с помощью их подстановки в это уравнение.

2. Является ли многочлен $x^4 + 4x^3 - 2x^2 - 12x + 9$ точным квадратом, т.е. *можно ли подобрать* три числа p, q и r так, чтобы имело место тождество $x^4 + 4x^3 - 2x^2 - 12x + 9 = (px^2 + qx + r)^2$?

3. *Убедитесь*, что многочлен $x^2 + y^2 + z^2$ нельзя представить в виде произведения $(ax + by + cz)(Ax + By + Cz)$ ни при каких вещественных или комплексных числах a, b, c, A, B, C.

4. *Проверьте*, что функция, являющаяся результатом вычисления выражения **Integrate[1/(2 + Cos[x]), x]**, не есть первообразная функции $1/(2 + \cos x)$. Напомним, что первообразной функции $f(x)$ называется такая непрерывная и дифференцируемая функция $F(x)$, что $F'(x) = f(x)$.

5. Неявная функция $y(x)$ определяется уравнением $xy^3 - (x + 1)^2 y + 3 = 0$ и условием $y(0) = 3$. *Найдите* значения первой и второй производной функции $y(x)$ при $x = 0$.

6. Функция $z(x, y)$ определяется как неявная функция уравнением $z^5 - xz + y^2 z^2 - 1 = 0$ и условием $z(0, 0) = 1$. *Найдите* значения первых производных этой функции в точке $(0, 0)$.

7. *Найдите* численно длину отрезка кривой $4y^2 = (x-1)^5$, заключенного внутри параболы $y^2 = x$. Используйте функцию **ImplicitPlot** директории Graphics и встроенную функцию **NIntegrate**.

8. Решение задачи Коши для следующей системы Лоренца трех обыкновенных дифференциальных уравнений

$$x'(t) = -3(x(t) - y(t)),$$
$$y'(t) = -x(t)z(t) + 30x(t) - y(t),$$
$$z'(t) = x(t)y(t) - z(t)$$

для промежутка времени t порядка 30 занимает слишком много времени, если решать эту задачу с помощью встроенной функции **NDSolve**. В пакете Programming Examples RungeKutta есть функция **RungeKutta**, которая значительно быстрее справляется с численным интегрированием систем обыкновенных дифференциальных уравнений. Первым аргументом этой функции является список правых частей автономных систем ОДУ, причем аргумент t у неизвестных функций явно не указывается. Вторым аргументом является список неизвестных функций, третьим аргументом — список начальных значений, в четвертом аргументе указывается интервал изменения независимой переменной и шаг численного интегрирования. Применительно к системе Лоренца обращение к функции **RungeKutta** может выглядеть следующим образом:

```
rk = RungeKutta[{-3(x - y), -x z + 30 x - y, x y - z},
{x, y, z}, {0, 1, 0}, {30, 0.05}];
```

В результате вычисления получается список rk значений неизвестных функций в точках разбиения интервала изменения независимой переменной. Этот список можно использовать для наглядного представления траектории.

```
Show[Graphics3D[{Line[rk]}], Axes -> True,
AxesLabel -> {"x", "y", "z"}]
```

Рисунок демонстрирует наличие так называемого „странного аттрактора“ у траекторий системы Лоренца.

Исследуйте, как изменяется рисунок в зависимости от изменений коэффициентов 3 и 30 в системе, а также от интервала и шага интегрирования. Сколько времени занимает численное интегрирование на вашем компьютере?

Глава 3
ВСТРОЕННАЯ ГРАФИКА

Трудно переоценить значение наглядных образов в научном и инженерном творчестве. Графики и рисунки служат прекрасным средством для лучшего понимания особенностей поведения математических объектов, которые часто остаются скрытыми от исследователя в формульном или тем более численном представлении. „Математика“ позволяет строить двух- и трехмерные графики функций и массивов численных данных, контурные и плотностные графики функций от двух аргументов, гистограммы, круговые диаграммы и т.п. В этой главе приводится описание функций, служащих для графического представления математических объектов и данных, а также более мощные средства визуализации, каковыми являются графические примитивы.

3.1. Графические функции и их опции

Встроенные функции, служащие для создания графиков, оканчиваются на **Plot** в двухмерном случае или на **Plot3D** — в трехмерном. Следовательно, их точные имена могут быть получены с помощью команд **?*Plot** или **?*Plot3D**.

ContourPlot	ListPlot
DensityPlot	ParametricPlot
ListContourPlot	Plot
ListDensityPlot	ParametricPlot3D
ListPlot3D	Plot3D

Все эти функции, кроме аргументов, значения которых обязательно указываются пользователем при обращении к функции, имеют большое количество аргументов необязательных,

или *опций*. Значения опций установлены по умолчанию, но при желании могут быть изменены пользователем. Первым обязательным аргументом является выражение или список выражений „Математики", которые будут интерпретироваться как функции, задающие линии или поверхности. Второй аргумент, называемый *итератор*, определяет, какие символы понимаются как аргументы функций и в каких пределах они изменяются. В трехмерном случае указываются два итератора. Опции определяют стиль и дополнительные элементы графических рисунков и позволяют добиться максимальной информативности и выразительности.

Следующие 20 опций являются общими для всех семи графических функций, рисующих двумерные объекты:

AspectRatio	ColourOutput	Frame	PlotRange
Axes	DefaultColour	FrameLabel	PlotRegion
AxesLabel	DefaultFont	FrameStyle	Prolog
AxesStyle	DisplayFunction	FrameTicks	RotateLabel
Background	Epilog	PlotLabel	Ticks

Опции задаются в виде правил подстановок, например AspectRatio → Automatic. По определению, AspectRatio есть отношение высоты к ширине двухмерного рисунка. Пользователь может установить его равным любому числу k, впечатав в качестве третьего аргумента функции выражение AspectRatio → k. По умолчанию, т.е. если опция пользователем специально не указана, это число равно $1/GoldenRatio$, где $GoldenRatio$, „золотое сечение", приближенно равно 1.61803. Установленная на Automatic опция AspectRatio будет определяться алгоритмами „Математики".

Опция Axes определяет, какие из координатных осей будут нарисованы. Возможны три установки: False, при которой ни одна из осей не будет представлена, True — обе оси нарисованы и $\{Boolean, Boolean\}$, где Boolean принимает значения True или False.

Опция AxesLabel позволяет дать названия осям. Например, $\{"x","y"\}$ или $\{"Time","Signal"\}$ и т.д.

При желании название всему рисунку можно дать с помощью опции PlotLabel в виде *PlotLabel* → "название".

Опция AxesStyle формирует способ представления осей координат путем задания графических директив, среди которых упомянем Thickness[d], определяющую относительную толщину осей, GrayLevel[d], определяющую насыщенность линии черным цветом, RGBColor[d_1, d_2, d_3] — цвет линии, Hue[d_1, d_2, d_3] — цвет линии в других цветовых координатах и установку Dashing[{$d1, d2, \ldots$}], определяющую размеры последовательных сегментов прерывистой линии (размеры повторяются циклически). Числа d, d_1, d_2 и d_3 заключены между 0 и 1. Пример задания опции AxesStyle:

AxesStyle → {{Thickness[0.007], RGBColor[1,0,0]},
{Thickness[0.008], Hue[0.65]}}

В рассматриваемом случае ось Ox будет начерчена красным цветом и толщиной, составляющей 0.007-ю часть общей ширины рисунка, в то время как ось Oy будет начерчена синим цветом и с относительной толщиной 0.008.

Опция Background определяет цвет фона рисунка. По умолчанию устанавливается на Automatic, что приводит к белому фону на большинстве дисплеев и принтеров. При установке пользователем цвет фона задается в одной из трех систем. Для дисплеев — в RGB[d_1, d_2, d_3] (красный, зеленый, синий) или в Hue[h, s, b], для цветных принтеров — в CMY[d_1, d_2, d_3, d_4] (синий, пурпурный, желтый, черный) системах. Здесь d_1, d_2, d_3 или d_1, d_2, d_3, d_4 есть относительные интенсивности базовых цветов в RGB или CMY системах, $h \in [0,1]$ — обязательный параметр, определяющий положение цвета в цветовой Hue-палитре, $s \in [0,1]$ — насыщенность, $b \in [0,1]$ — яркость цвета. Последние параметры не являются обязательными и по умолчанию полагаются равными 1. Возможно также задание фона в виде Background → GrayLevel[d], т.е. серый цвет фона относительной интенсивности $d \in [0,1]$ ($d = 0$ — черный фон, $d = 1$ — белый фон).

При построении рисунков „Математика“ воспроизводит цвета соответственно директивам, которые даны пользователем или устанавливаются по умолчанию.

Опция ColorOutput позволяет преобразовать эти директивы с тем, чтобы приспособить выходные данные к конкретному графическому устройству. Например, установка ColorOutput → GrayLevel преобразует все цвета к оттенкам серого.

Опция DefaultColor определяет цвета элементов рисунка: точек, линий, надписей и т.п. По умолчанию установлена на Automatic, что обеспечивает цвет элементов рисунка, дополнительный к цвету фона. Белые координатные оси и линии графиков на черном фоне получаются при установках Background → GrayLevel[0] и DefaulColor → GrayLevel[1].

Опция DefaultFont позволяет установить тип и размер шрифта для текста, включенного в рисунок, и для масштабных отметок на осях, если установленные по умолчанию шрифт или его размер пользователя почему-либо не устраивают.

Результатами вычислений, проведенных двумерными графическими функциями, служат помещаемые в выходную ячейку выражения „Математики“ с заголовками Graphics, DensityGraphics или ContourGraphics. Сам рисунок есть как бы побочный эффект вычислений. Бывают ситуации, когда появление рисунка нежелательно. В этом случае следует воспользоваться опцией DisplayFunction → Identity. Опции Epilog и Prolog будут описаны в п. 7.

С помощью опции Frame → True рисунок заключается в прямоугольную рамку, четыре стороны которой можно снабдить надписями, сделав установку FrameLabel → {"label1","label2", "label3","label4"}. Надписи label располагаются по часовой стрелке, причем label1 помещается внизу рисунка. Надписи label2 и label4 по умолчанию располагаются вертикально и читаются снизу вверх, но опция RotateLabel → False располагает

их горизонтально. Установка FrameLabel → "label" помещает единственную надпись около левой стороны рамки.

Опция FrameStyle аналогична опции AxesStyle. При наличии рамки координатные отметки ставятся не на осях координат, а на нижней и левой сторонах рамки. Эти отметки ставятся в соответствии с алгоритмами „Математики" в случае FrameTicks → Automatic. Если пользователь хочет расставить их сам, то ему нужно сделать установку FrameTicks → $\{\{x_1, x_2, \ldots\}, \{y_1, y_2, \ldots\}\}$, где x_i, y_j — координаты отметок. Эти же символы или числа будут поставлены рядом с точками с координатами x_i, y_j. Одну из двух внутренних фигурных скобок с координатами отметок можно заменить на Automatic. В случае когда желательны другие надписи вблизи координатных отметок или желательно отсутствие всяких надписей, то следует прибегнуть к установке FrameTicks → $\{\{\{x_1, xlabel1\}, \{x_2, xlabel2\}, \ldots\}, \{\{y_1, ylabel1\}, \{y_2, ylabel2\}, \ldots\}\}$. При xlabel или ylabel, равных " ", надписи вблизи соответствующих координатных меток будут отсутствовать.

Приступая к описанию опций PlotRange и PlotRegion, отметим, что выполнение графических вычислений проходит три стадии. На первой „Математика" формирует последовательность графических примитивов — точек, линий, многоугольников и т.п., из которых состоит рисунок. На второй — вырабатывается описание построенной последовательности примитивов на языке PostScript. Это описание делает рисунок независимым от компьютерных платформ или графических устройств. Оно допускает возможность преобразований рисунка и его непосредственное воплощение на лазерных принтерах и других устройствах. Именно это описание является формальным результатом графических вычислений и помещается в выходную ячейку.

Наконец, на третьей стадии это описание транслируется применительно к графическому устройству, на котором работает пользователь. Вычисленный графический рисунок

помещается в отведенное ему на устройстве пространство, заполняя собой все это пространство при условии, что опция PlotRegion по умолчанию установлена на $\{\{0,1\},\{0,1\}\}$, где первое $\{0,1\}$ есть область изменения нормированной координаты x, второе — область изменения нормированной координаты y. Таким образом, отведенное пространство в нормированных координатах представляет собой единичный квадрат. Если же произведена установка PlotRegion→ $\{\{sxmin, sxmax\},\{symin, symax\}\}$, где sx, sy — нормированные x и y координаты, то рисунок будет заполнять соответствующую часть пространства, т.е. вокруг рисунка возникнут поля.

В процессе вычислений, относящихся к первой стадии, „Математика" определяет, в частности, какая область изменения координаты y должна быть отображена на рисунке с тем, чтобы рисунок включал наиболее интересные черты графического объекта. При этом „нехарактерные" значения и очень большие значения координаты y исключаются при установке по умолчанию опции PlotRange → Automatic. Другие возможные установки этой опции есть All (отображаются все значения), $\{ymin, ymax\}$ (на оси Oy представлены значения функций, заключенные в области от $ymin$ до $ymax$) и $\{xrange, yrange\}$ (отображаются только те части графика, x-координата которых заключена в области $xrange$, а y-координата — в области $yrange$).

Последняя опция Ticks, управляющая координатными отметками на координатных осях, аналогична ранее описанной опции FrameTicks.

Выше приведены опции, общие для всех двумерных графических функций. Каждая из функций, рассмотренных ниже, имеет свои дополнительные опции. Последние, а также некоторые опции трехмерных графических функции будут даны в описаниях соответствующих функций. В заключение этого пункта дадим пример применения опций для графической функции **Plot** (рис. 3.1):

```
pl1 = Plot[Sin[x^2], {x, -Pi, Pi},
PlotStyle → Thickness[0.005],
AxesStyle → {Thickness[0.004]},
AxesLabel → {"x", "y"},
GridLines → Automatic,
PlotLabel → "Plot of Sin[x^2]",
DefaultFont → {"Arial", 14}];
```

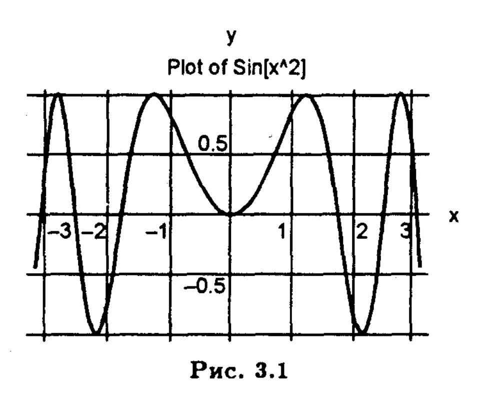


Рис. 3.1

В этом примере заданы в явном виде далеко не все опции функции **Plot**. Какие именно опции фактически использованы и какие значения установлены для этих опций, можно узнать с помощью функции **Options**, вычислив выражение **Options**[pl1].

3.2. Двумерная графика

Начнем описание с важнейшей и часто употребляемой функции **Plot**, тем более что она уже встречалась нам ранее. С помощью этой функции можно чертить графики сразу нескольких функций, задав в качестве первого аргумента функции **Plot** список функций. Для того чтобы различить графики, используется опция PlotStyle → $\{\{dir_1\}, \{dir_2\}, \ldots\}$, где dir_i есть совокупность графических директив, относящихся к i-му графику (рис. 3.2):

```
pl2 = Plot[{-40x^2 + 800x + 30, 100x + 3000}, {x, 6, 12},
PlotStyle → {{Thickness[0.008]},
{Thickness[0.006], Hue[0.]}}]
```

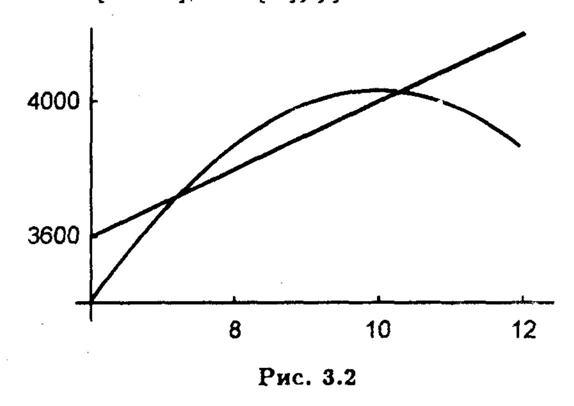


Рис. 3.2

На цветном мониторе график параболы будет нарисован с помощью толстой черной линии, а график прямой — с помощью более тонкой красной. Мы видим, что координатные оси пересекаются на этом рисунке не в точке (0,0). Опция AxesOrigin → $\{xorigin, yorigin\}$ определяет пересечение осей в точке с координатами $(xorigin, yorigin)$.

При построении графиков функций возможны две стратегии. Первая состоит в том, чтобы сначала получить возможно простое приближенное представление для функции в заданном интервале и вычислять затем приближенные значения в каком-то наборе точек. Вторая стратегия, которой обычно следует „Математика", — определить заранее характерные точки графика и затем вычислять функцию в этих точках. Тем не менее существуют ситуации, когда целесообразно заставить „Математику" следовать первой стратегии.

Один из таких случаев реализуется при построении нескольких графиков одновременно и тогда, когда соответствующие функции заданы не пользователем, а получаются при вычислении функции типа **Table**. Второй случай — построение графиков функций, полученных с помощью **NDSolve**. В подобных случаях прибегают к функции **Evaluate**. В рассматриваемом

ниже примере строятся графики первых четырех полиномов Чебышева первого рода: ChebyshevT[0, x] — полинома нулевого порядка, равного тождественно единице, Chebyshev$T$[1, x] = x, а также полиномов ChebyshevT[2, x], ChebyshevT[3, x] (рис. 3.3):

```
pl3 = Plot[Evaluate[Table[ChebyshevT[n,x], {n,0,3}]],
{x, -1, 1},
PlotStyle→Table[{Thickness[0.003+2i 10^-3]},{i,0,3}]];
```

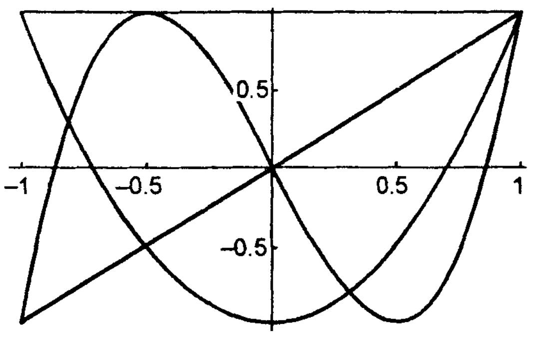


Рис. 3.3

Рассмотрим теперь остальные из семи специфических для функции **Plot** опций: AxesOrigin, Compiled, GridLines, MaxBend, PlotDivision, PlotPoints, PlotStyle.

Опция GridLines, установленная на Automatic, вызывает появление на рисунке сетки вертикальных и горизонтальных прямых, исходящих из точек координатных осей с координатными отметками. Эта сетка видна на графике pl1, приведенном выше.

Опция PlotStyle совпадает с опциями AxesStyle и FrameStyle, но определяет графический стиль представления вычерчиваемых линий.

Адаптивный алгоритм, которому следует „Математика“ при построении графиков, предусматривает получение достаточно гладких кривых. Одним из критериев гладкости является величина угла между последовательными отрезками пря-

мых, аппроксимирующих кривую между соседними точками, в которых вычисляются значения функций, задающих кривую. Величина этого угла устанавливается опцией MaxBend, по умолчанию он равен 10°.

Количество точек, в которых производятся вычисления, регулируется опцией PlotPoints, которая, если не оговорено иное, равна 25. Если угол между последовательными сегментами превосходит значение опции MaxBend, то производится дробление интервала между соседними точками, но таких шагов дробления не может быть больше, чем значение опции PlotDivision. Для уменьшения времени вычисления функций в выбранных по адаптивному алгоритму точках производится их компилирование, заключающееся в создании некоторого псевдокода. Однако последний может привести к уменьшению точности вычислений. Если нужно избежать подобной потери точности, то следует прибегнуть к опции Compiled → False.

Функция **ParametricPlot** позволяет рисовать кривые и семейства кривых, заданных параметрически. Ее описание с соответствующими изменениями совпадает с описанием функции **Plot**, в частности она имеет те же опции. Мы ограничимся поэтому простым примером применения этой функции (рис. 3.4):

pl4 = ParametricPlot[{Cos[t], Sin[3t]}, {t, 0, 2Pi},
PlotLabel → "Lissajou figure"]

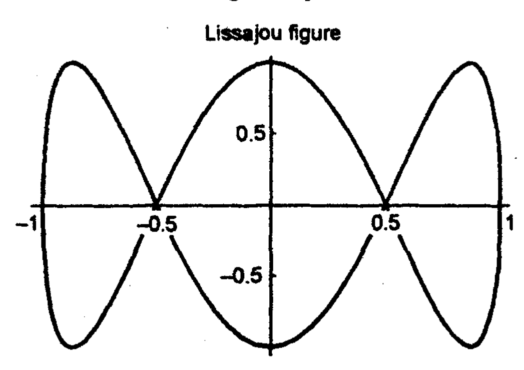


Рис. 3.4

Функция **ListPlot** предназначена для графического представления дискретных численных данных и имеет своим первым аргументом либо список вида data1=$\{y_1, y_2, \ldots y_n\}$, либо список вида data2=$\{\{x_1, y_1\}, \{x_2, y_2\}, \ldots \{x_n, y_n\}\}$, либо список данных создается какой-либо функцией, например **Table**. В первом случае будет графически представлена совокупность точек с координатами (i, y_i), во втором — с координатами (x_i, y_i). Пусть массив данных data3 имеет вид:

data3 = {1., 1.18785, 0.81045, 0.09574, −0.41614, −0.51789, −0.49499, −0.60536, −0.65364, −0.28532, 0.42549, 0.95922, 0.96017}

Ниже следует его графическое представление (рис. 3.5):

ListPlot[data3, PlotStyle → {Pointsize[0.04]}]

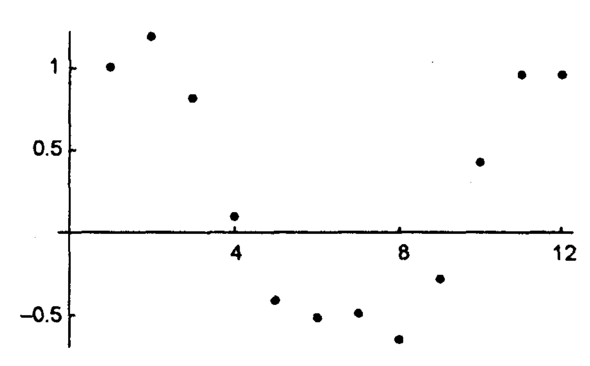


Рис. 3.5

Следует обратить внимание, что графическая директива, управляющая размером точек, не Thickness, как это было в случае линий, а PointSize. Функция **ListPlot** имеет дополнительные опции AxesOrigin, GridLines, PlotJoined, PlotStyle. С помощью PlotJoined → True точки на графике будут соединены отрезками прямых линий (рис. 3.6):

ListPlot[data3, PlotStyle → {Pointsize[0.04]}, PlotJoined → True]

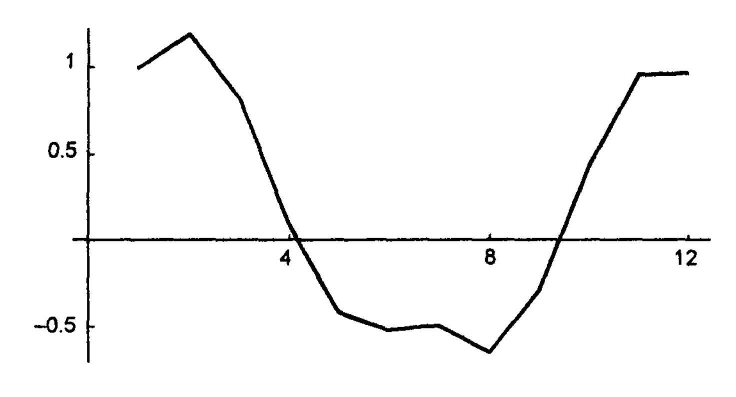


Рис. 3.6

Функции **ContourPlot** и **DensityPlot** позволяют графически изучать поведение функций от двух независимых переменных. Первая из них строит линии уровня, являющиеся линиями на плоскости независимых переменных, где функция принимает постоянные значения. Первым аргументом рассматриваемых функций является выражение „Математики", задающее функцию от двух переменных. Какие из символов в выражении следует понимать как независимые переменные, определяется с помощью двух итераторов вида $\{var_i, varmin_i, varmax_i\}$, следующих за первым аргументом.

Дополнительными опциями функции **ContourPlot** являются опции AxesOrigin, ColorFunction, Compiled, ContourLines, Contours, ContourShading, ContourSmoothing, ContourStyle, PlotPoints. Рисунок с линиями уровня по умолчанию заштриховывается или раскрашивается. В случае опции ColorFunction → Automatic области, где сосредоточены линии уровня с меньшими значениями функции, имеют более темные оттенки серого цвета, чем области с большими значениями функции (рис. 3.7).

ContourPlot[Cos[x^2]Sin[y], {x, -2.2, 2.2}, {y, -3, 3}]

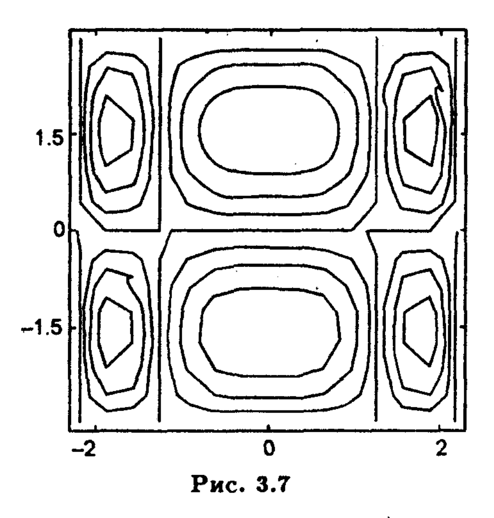


Рис. 3.7

На рисунке ясно видны минимумы и максимумы функции в рассматриваемой области.

При установке опции ColorFunction → Hue происходит цветное раскрашивание, что делает рисунок более выразительным, но имеет тот недостаток, что в силу периодичности цветовой Hue-палитры минимумы и максимумы функции раскрашены в один и тот же красный цвет.

Впрочем, при установке ContourShading → False раскрашивание отсутствует, а если установить опцию ContourLines → False, то отсутствуют линии уровня. По умолчанию число линий уровня на рисунке равно десяти, пользователь может установить необходимое число с помощью опции Contours.

При установке ContourSmoothing → True происходит сглаживание линий уровня с тем, чтобы рисунок имел достаточно эстетичный вид.

Функция DensityPlot отображает значения исследуемой функции в ячейках регулярной сетки с помощью окрашивания этих ячеек либо в серый цвет, либо в цвета Hue-палитры. Дополнительные опции функции **DensityPlot** есть

{ColorFunction, Compiled, Mesh, MeshStyle, PlotPoints}. Сетку ячеек можно исключить из рисунка, установив Mesh → False. Функции **ListContourPlot** и **ListDensityPlot** аналогичны только что рассмотренным двум последним функциям и строят соответствующие графические объекты для дискретных данных.

3.3. Трехмерная графика

Функции **Plot3D**, **ParametricPlot3D** и **ListPlot3D** создают трехмерные графические объекты (рис. 3.8):

Plot3D[Cos[x^2]Sin[y],{x,-2.2,2.2},{y,-3,3}]

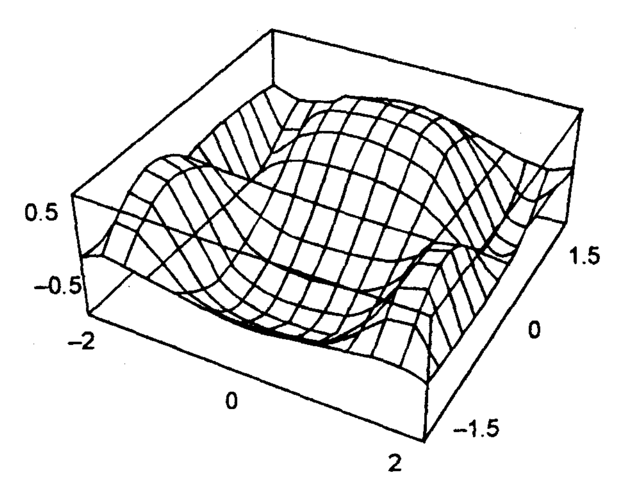


Рис. 3.8

Характерной чертой трехмерного стиля является заключение графика функции в коробочку, от которой можно избавиться с помощью опции Boxed → False. Опишем кратко общие опции трехмерных графических функций, за исключением тех, которые использовались в двухмерной графике. Вот полный список этих общих опций.

AmbientLight	Boxed	Epilog	Prolog
AspectRatio	BoxRatios	FaceGrids	Shading
Axes	BoxStyle	Lighting	SphericalRegion
AxesEdge	ColorOutput	LightSources	Ticks
AxesLabel	DefaultColor	PlotLabel	ViewCenter
AxesStyle	DefaultFont	PlotRange	ViewPoint
Background	DisplayFunction	PlotRegion	ViewVertical

Опции AspectRatio, Axes, AxesLabel, AxesStyle, Background, ColorOutput, DefaultColor, DisplayFunction, Epilog, PlotLabel, PlotRange, PlotRegion и Ticks совпадают с соответствующими опциями двухмерных графических функций.

Опция Shading устанавливает или отменяет закрашивание поверхности в серый или иные цвета в зависимости от расположения соответствующего участка поверхности. Если Shading → False, то представление о поверхности можно получить по имеющейся на ней криволинейной сетке, образованной линиями пересечения плоскостей $x = \text{const}$, $y = \text{const}$ с поверхностью. В случае Shading → True окраска поверхности определяется дополнительными опциями, содержащими символ Light.

Опция Lighting определяет, будет ли использована искусственная подсветка двухмерной поверхности. Если установлено Lighting → True, то характер подсветки определяется опциями AmbientLight (фоновый цвет) и LightSources (источники света).

С помощью опции AmbientLight → GrayLevel[d] осуществляется равномерное окрашивание поверхности в серый цвет или при AmbientLight → Hue[d] в один из цветов Hue-палитры. В то же время может осуществляться подсветка поверхности красным светом из источника, расположенного в точке с координатами $\{x_r, y_r, z_r\}$, зеленым светом из точки $\{x_g, y_g, z_g\}$ и синим светом из точки $\{x_b, y_b, z_b\}$. По умолчанию сделаны установки AmbientLight → GrayLevel[0] и LightSources → → {{{1.,0.,1.}, RGBColor[1,0,0]}, {{1.,1.,1.}, RGBColor[0,1,0]}, {{0.,1.,1.}, RGBColor[0,0,1]}}].

При установке Lighting → False поверхность подкрашивается серым цветом более темным на участках с меньшими значениями z-координаты.

Опция ViewPoint определяет точку, из которой рассматривается рисуемая поверхность. Положение этой точки определяется относительными координатами, в которых центр коробочки с рисунком имеет координаты $\{0,0,0\}$, а линейный размер ее наибольшей стороны не превосходит единицы. Относительные размеры других сторон определяются опцией BoxRatios. По умолчанию сделана установка ViewPoint → $\{1.3,-2.4,2\}$. Другие часто используемые опции: $\{0,-2,0\}$ — взгляд на фасад коробочки, $\{0,-2,2\}$ — сверху с фасадной стороны, $\{0,-2,-2\}$ — снизу с фасадной стороны, $\{-2,-2,0\}$ — взгляд с левого угла, $\{2,-2,0\}$ — взгляд с правого угла, $\{0,0,2\}$ — строго сверху.

Опции ViewCenter и ViewVertical определяют положение коробочки в поле, отведенном для рисунка.

Пример применения функции **Plot3D** был приведен раньше. Опцией, специфической для этой функции, является Mesh. При установке Mesh → False убирается криволинейная сетка линий на поверхности.

Опция ColorFunction управляет интенсивностью и цветом подкраски поверхности, зависящими от значения координаты z. По умолчанию она установлена на Automatic.

Опция ClipFill определяет, будут ли нарисованы плоскости $z = zmax$ или $z = zmin$ в случае, когда значение функции в какой-то области превосходит значение опции PlotRange.

Опция HiddenSurface в случае ее установки по умолчанию на True обеспечивает подсветку и подкрашивание всей поверхности как сплошного тела, в том числе и невидимых с выбранной точки зрения участков поверхности. В случае HiddenSurface → False подкрашивание вовсе отсутствует.

С помощью функции **ParamericPlot3D** можно рисовать параметрически заданные двухмерные поверхности или одномерные кривые в трехмерном пространстве. Ниже следуют рисунки тора и одномерной кривой (рис. 3.9 и 3.10):

**ParametricPlot3D[{(2 – Cos[u])Cos[t],
(2 – Cos[u])Sin[t], Sin[u]}, {t, 0, 2Pi}, {u, 0, 2Pi},
Boxed → False, Axes → False];**

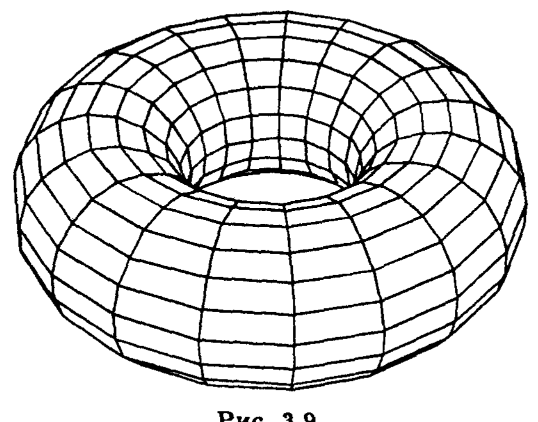


Рис. 3.9

ParametricPlot3D[{Cos[2t], Sin[2t], Sin[t]}, {t, 0, 2Pi}];

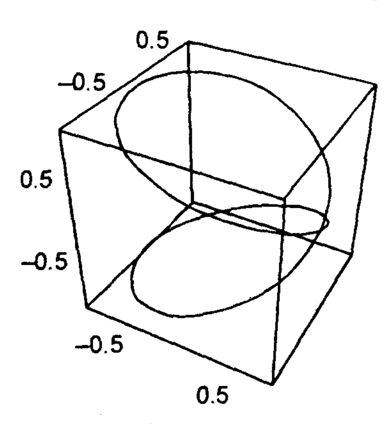


Рис. 3.10

Функция **ListPlot3D** служит для графического представления дискретных численных данных вида data = $\{\{z_{11}, z_{12}, \ldots\}, \{z_{21}, z_{22}, \ldots\}, \ldots\}$. Данные data интерпретируются как z-координаты трехмерных точек, причем значение координаты z_{ij}

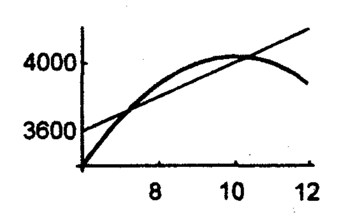
приписывается точке с координатами $x = i$, $y = j$. Через эту совокупность трехмерных точек проводится двухмерная поверхность, аналогично тому как двумерные точки в функции **ListPlot** соединялись ломаной в случае установки опции PlotJoined → True.

3.4. Изменение стиля и комбинирование построенных рисунков

Пусть выполнен какой-то рисунок или серия рисунков. Может случиться, что требуется изменить графический стиль рисунка, объединить несколько графиков в одном рисунке или представить совместно несколько рисунков. Для этого не нужно вычислять рисунки заново, а следует прибегнуть к функции **Show**. Выражение **Show[plot]** заново рисует уже вычисленный рисунок **plot**. При этом можно изменить установки опций в той графической функции, с помощью которой был получен рисунок, что легко сделать с помощью **Show[plot, option → value]**. Выражение **Show[plot$_1$, plot$_2$, ...]** при вычислении объединяет несколько графиков в одном. Согласование областей задания и областей значения функций происходит автоматически.

Поместить несколько рисунков рядом в одной линии, вертикальной или горизонтальной, а также объединить их в прямоугольную таблицу можно следующим образом (рис. 3.11, 3.12 и 3.13):

Show[GraphicsArray[{pl2, pl3}]];



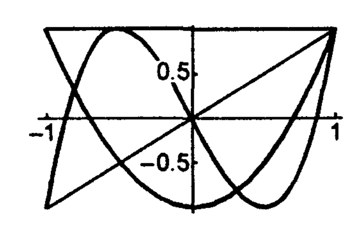


Рис. 3.11

Show[GraphicsArray[{{pl3},{pl4}}]];

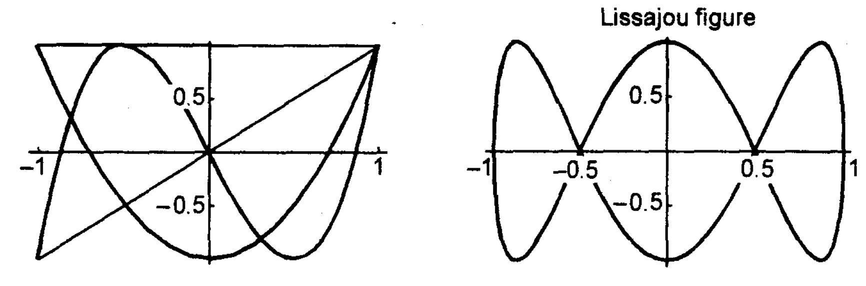


Рис. 3.12

Show[GraphicsArray[{{pl2,pl3},{pl4,pl6}}]];

Lissajou figure

Рис. 3.13

3.5. Мультипликация

„Математика" позволяет проводить визуализацию динамических процессов. Если создать серию графических рисунков, то они могут быть оживлены подобно тому, как серия неподвижных картинок-кадров соединяется в мультфильм. Технологию

создания мультфильма продемонстрируем на простом двумерном примере: бегущей синусоидальной волне.

Сначала сделаем серию из 11 рисунков, каждый из которых представляет собой график функции **Sin[x – t]** на отрезке оси x от 0 до 4π для фиксированных $t = 0, 2\pi/10, 4\pi/10, \ldots, 2\pi$. Эти рисунки будут помещены каждый в свою ячейку „Записной книжки", расположенные одна под другой:

Table[Plot[Sin[x – t], {x, 0, 2Pi}], {t, 0, 2Pi, 2Pi/10}];

Выделим все эти 11 ячеек и затем с помощью команды **Graph Animate Selected Graphics** та из ячеек, которая полностью представлена на экране дисплея, будет оживлена. Внизу экрана при этом появляются шесть кнопок, позволяющие изменить направление движения анимации, ускорить, замедлить ее или сделать паузу.

При подготовке рисунков к анимации следует проявить известную предусмотрительность, особенно полезную при оживлении трехмерных рисунков. Так, например, область значения функций для всех графиков должна быть одна и та же, а не выбираться автоматически. Поэтому нужно установить соответствующим образом опцию PlotRange. Для трехмерных рисунков следует также установить опцию SphericalRegion → True, обеспечивающую неподвижность коробочки с графическим объектом от рисунка к рисунку.

3.6. Графические функции специализированных пакетов

Пакет Graphics — один из самых интересных специализированных пакетов, содержащий около пятидесяти графических функций. Рассмотрим некоторые из них. Функции **ContourPlot3D** и **ListContourPlot3D** являются аналогами соответствующих встроенных двумерных функций **ContourPlot** и **ListContourPlot**. Мы уже пользовались функцией **ImplicitPlot** при решении алгебраических и трансцендентных уравнений. Функция

FilledPlot очень полезна для решения уравнений и неравенств (рис. 3.14):

FilledPlot[{Sin[x],0.1x},{x,0,4Pi},Fills → {{1,2}, GrayLevel[0.9]}];

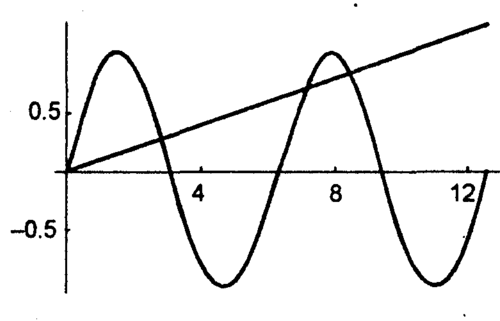


Рис. 3.14

На экране дисплея закрашена серым цветом область, заключенная между графиками функций **Sin[x]** и **0.1x**. В общем случае графиков может быть несколько, закрашиваться своим особым цветом могут области, заключенные между любыми парами графиков, а также между каким-то графиком и осью Ox. Функция **FilledPlot** имеет дискретный аналог **ListFilledPlot** (рис. 3.15):

ListFilledPlot[{2,3,2,2,5,3}];

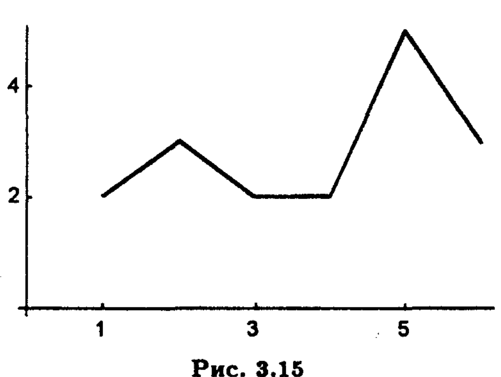


Рис. 3.15

Часто кривые на плоскости задаются в полярных координатах. Если воспользоваться функцией **PolarPlot**, то их не нужно пересчитывать в декартовы координаты для функции **Plot**.

Гистограммы для дискретных данных получаются с помощью нескольких функций, являющихся модификациями функции **BarChart**. Пусть даны два массива данных $data1 = \{2,3,2,2,5,3\}$ и $data2 = \{3,2,1,5,6,3\}$, тогда их гистограмма (рис. 3.16) может быть получена вычислением выражения

BarChart[data1, data2]

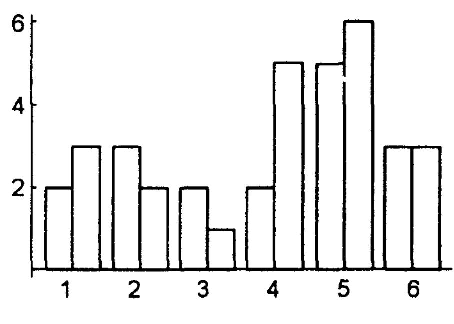


Рис. 3.16

Функция **PieChart** рисует круговые гистограммы данных (рис. 3.17):

PieChart[data1]

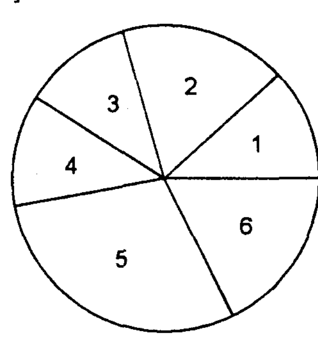


Рис. 3.17

3.7. Графические примитивы

Двух- и трехмерные рисунки, которые выполняются графическими функциями, состоят из графических примитивов, таких, как точки, линии, многоугольники, круги, диски, параллелепипеды, текст. Точка двумерного рисунка как графический примитив есть выражение вида **Point**[{**x**,**y**}], где x и y суть декартовы координаты точки на плоскости. Линия есть на самом деле ломаная линия, состоящая из отрезков прямых, соединяющих последовательные угловые точки этой линии: **Line**[{{$\mathbf{x_1}$,$\mathbf{y_1}$},{$\mathbf{x_2}$,$\mathbf{y_2}$},...}]. Рассмотрим два графических примитива:

p = **Point**[{**1**,**1**}]

и

l = **Line**[{{**0**,**0**},{**1/2**,**2**},{**3/2**,**2**},{**2**,**0**}}]

Превратить их в рисунок на экране дисплея можно следующим способом. Сначала к ним нужно применить функцию **Graphics**, трансформируя их в один из шести графических объектов (Graphics, DensityGraphics, ContourGraphics, Graphics3D, SurfaceGraphics и GraphicsArray), а к результату применить функцию **Show** (рис. 3.18):

Show[**Graphics**[{**p**,**l**}]];

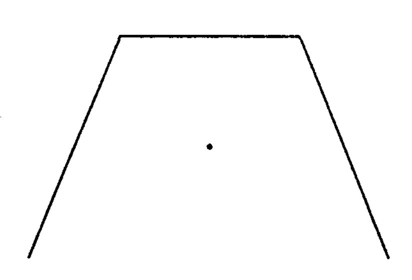


Рис. 3.18

Кроме графических примитивов имеются графические директивы, определяющие размеры, цвет и стиль представления примитивов на экране. Если они не указаны явно, как в только что рассмотренном примере, то они устанавливаются по умолчанию. Директивы PointSize[r] и AbsolutePointSize[r] определяют относительный и абсолютный размеры точки как круга радиуса r. В первом случае r есть отношение радиуса к ширине полного рисунка, во втором случае — r есть радиус в единице длины, приблизительно равной одной семьдесят второй дюйма. Директива ставится перед примитивом, и оба эти объекта заключаются в фигурные скобки. Кроме того, директива может действовать на несколько однородных примитивов, стоящих за ней. Вся группа также заключается в фигурные скобки. Графических директив может быть несколько, и они должны предшествовать примитивам. Вычисление выражения

Show[Graphics[{Pointsize[0.03],Hue[0],
Table[Point[{0.1j,0.05j}],{j,5}]}]];

приводит к появлению на экране пяти точек красного цвета относительного размеры 0.03. Директивами, управляющими представлением линии, кроме цвета, являются Thickness[r] и AbsoluteThickness[r]. Они определяют относительную и абсолютную ширину линии.

Директива Dashing[{r1,r2,...}] имеет результатом представление линии в виде совокупности отрезков длины r1, r2 и т.д., которые повторяются циклически. Пусть l есть следующий примитив:

l = **Line[{{0,0},{1,1},{0.95,0.98},{1,1},{0.98,0.95}}]**

тогда можно получить следующий рисунок (рис. 3.19):

Show[Graphics[{Dashing[{0.05,0.03}], Thickness[0.01],l}]];

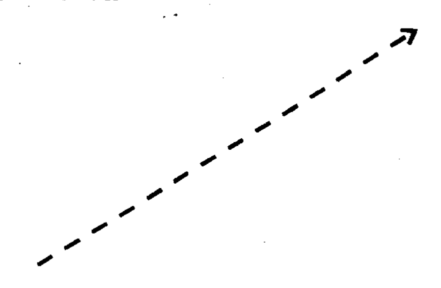


Рис. 3.19

Графический примитив **Polygon**[{**x1**, **y1**}, {**x2**, **y2**}, ...}] представляет заполненный многоугольник, ограниченный замкнутой ломаной линий, проходящей через точки $(x1, y1)$, $(x2, y2)$ и т.д. (рис. 3.20):

Show[Graphics[{Hue[0],Polygon[{{0,0},{1,1},{2,0}}]}]];

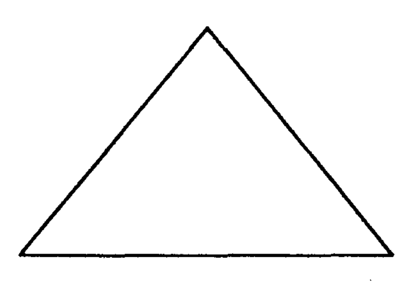


Рис. 3.20

На цветном мониторе получается заполненный красной краской равносторонний треугольник. Частный случай многоугольника — прямоугольник — можно получить с помощью примитива **Rectangle** [{{**xmin**, **ymin**}, {**xmax**, **ymax**}], в котором $(xmin, ymin)$ — координаты нижнего, а $(xmax, ymax)$ — координаты верхнего углов прямоугольника. Дугу окружности рисуют, прибегая к графическому примитиву **Circle**[{**x**,**y**}, **r**, {θ_1, θ_2}] и указывая координаты (x, y) центра окружности,

радиус r и граничные значения (θ_1, θ_2) полярного угла. Закрашенный круг получается с помощью примитива **Disk[{x,y},r]**.

Важным графическим примитивом является тот, который задает текст: **Text[expr,{x,y}]**.

В этом выражении *expr* есть любое выражение, и печатная форма вычисленного выражения будет центрирована относительно точки с координатами x, y. Если сначала нарисовать график параболы $y = (2+x)^2$ на отрезке $[-3,-1]$ и обозначить через p графический объект $p =$ **Plot[(2+x)^2,{x,−3,−1}]**, то этот рисунок можно снабдить текстом (рис. 3.21):

Show[p,Graphics[{Text["Graph of y="Expand[(2+x)^2], {−2,0.9}]}]];

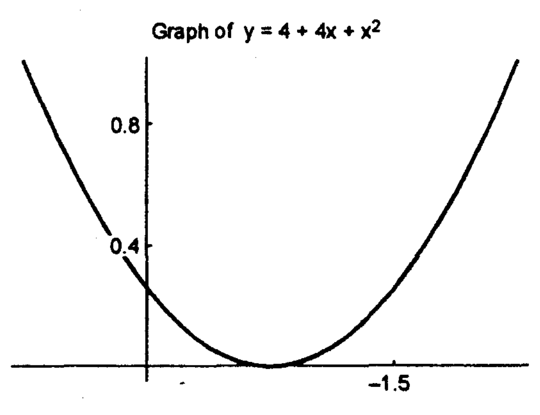


Рис. 3.21

Внести текст или другие графические примитивы в рисунки, созданные встроенными графическими функциями, можно с помощью опций Prolog и Epilog. По умолчанию они установлены на пустой список { }. Пользователь может внести в этот список любые подходящие случаю примитивы. Разница между рассматриваемыми опциями состоит в том, что примитивы опции Prolog рисуются до, а примитивы опции Epilog — после основного рисунка (рис. 3.22):

Plot[{AiryAiPrime[x], AiryAi[x]}, {x, −5, 5},
Prolog → {Text[FontForm["AiryAiPrime",
{"Times", 14}], {1.4, −0.3}],
Text[FontForm["AiryAi", {"Eurostyle", 14}], {1, 0.4}]}
PlotLabel → FontForm["TwoGraphs", {"Courier", 14}],
PlotStyle → {{Dashing[0.02]}, {GrayLevel[0.8]}}]

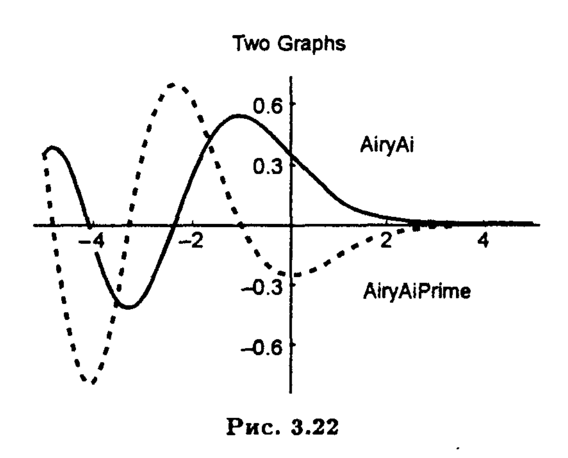


Рис. 3.22

В графическом примитиве Text была использована функция **FontForm**, аргументами которой являются строка с текстом, название и размер шрифта.

Если вместо заголовка Graphics воспользоваться заголовком Graphics3D, то можно получать трехмерные рисунки. При этом примитивы Point, Line, Text и Polygon с соответствующими изменениями переносятся на трехмерный случай. Единственным новым трехмерным примитивом является

Cuboid[{xmin, ymin, zmin}, {xmax, ymax, zmax}],

с помощью которого рисуется параллелепипед с противоположными углами в точках с координатами $xmin$, $ymin$, $zmin$ и $xmax$, $ymax$, $zmax$ (рис. 3.23):

```
Show[Graphics3D[{GrayLevel[0.8],Cuboid[{0,0,0},
{1,1,1}]}],
Boxed → False,Lighting → False];
```

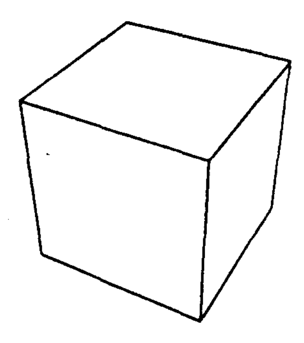


Рис. 3.23

Как и в двухмерном случае, рисунки, отвечающие разным примитивам, можно накладывать друг на друга (рис. 3.24):

```
Show[Graphics3D[{GrayLevel[0.9],Cuboid[{0,0,0},
{1,1,1}]}],
Graphics3D[{Hue[0],Thickness[0.01],
Line[{{-1,0,0},{2,1,1}}]}],
Graphics3D[{Hue[0.6],PointSize[0.04],
Point[{-1,0,0}]}],],
Graphics3D[{Hue[0.6],PointSize[0.04],Point[{2,1,1}]}],
Boxed → False,Lighting → False];
```

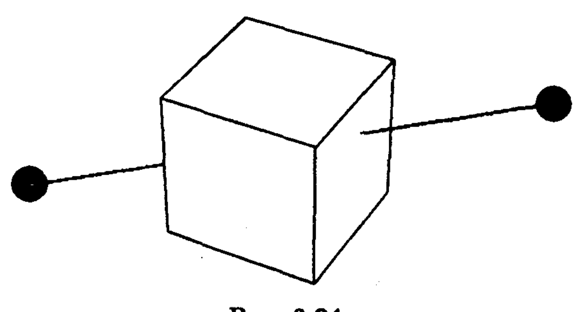


Рис. 3.24

Упражнения

1. Касательная как предельное положение секущих. *Постройте* совместные графики функции $f(x) = \sin x$, касательной к графику этой функции в точке 0.5 и пяти секущих, проходящих через точку $(0.5, f(0.5))$ и точки $(0.5 - j\ 0.04, f(0.5 - j\ 0.04))$, $j = 5, 4, 3, 2, 1$, на интервале от 0.01 до 1. *Окрасьте* график функции $f(x)$ в синий, касательную в красный и секущие в зеленый цвета. *Подберите* толщину кривых так, чтобы рисунок был наиболее выразителен. *Сделайте* мультфильм из серии рисунков, на каждом из которых нарисованы графики функции, касательной и одной из секущих.

2. *Постройте* совместные графики функции $f(x) = 5x^4 - 12x^3 + 8x^2 + 2$ и ее производной на интервале $(-1, 2)$. *Окрасьте* график функции в синий, а ее производной в красный цвета. *Сравните* интервалы убывания и возрастания функции со знаком ее производной.

3. Какой функцией из директории Graphics следует воспользоваться, чтобы нарисовать лемнискату $4(x^2 + y^2)^2 = 41(x^2 - y^2)$? Какие опции следует установить с тем, чтобы лемниската имела фиолетовый цвет и относительную толщину 0.01?

4. В рисунке Two Graphics *отметьте* графики функций AiryAiPrime(x) и AiryAi(x) вместо надписей маленькими квадратом и треугольником.

5. Формулы $x = \mathrm{ch}(u)\cos(t)$, $y = \mathrm{sh}(u)\sin(t)$ задают переход к эллиптической системе координат. Линии постоянного u суть эллипсы, линии постоянного t суть гиперболы. *Постройте* рисунок, содержащий пять линий постоянных u и пять линий постоянных t. *Используйте* для этого функции **ParametricPlot**, **Table**, **Show**.

6. *Подгрузите* пакет Graphics'Shapes', сделав доступными следующие геометрические объекты: **Cylinder** (цилиндр), **Cone** (конус), **Torus** (тор), **Sphere** (сфера), **MoebiusStrip** (лист Мебиуса), **Helix** (спираль) и **DoubleHelix** (двойная спираль). Что нужно сделать, чтобы нарисовать эти объекты?

7. *Подгрузите* пакет Graphics'Polyhedra' и *узнайте* названия многогранников, которые можно нарисовать с помощью функций этого пакета. Задав вопрос ?Cube, *узнайте*, что нужно сделать, чтобы нарисовать их с помощью функции **Show**. *Нарисуйте* эти объекты.

8. ***Нарисуйте*** картинки (рис. 3.25):

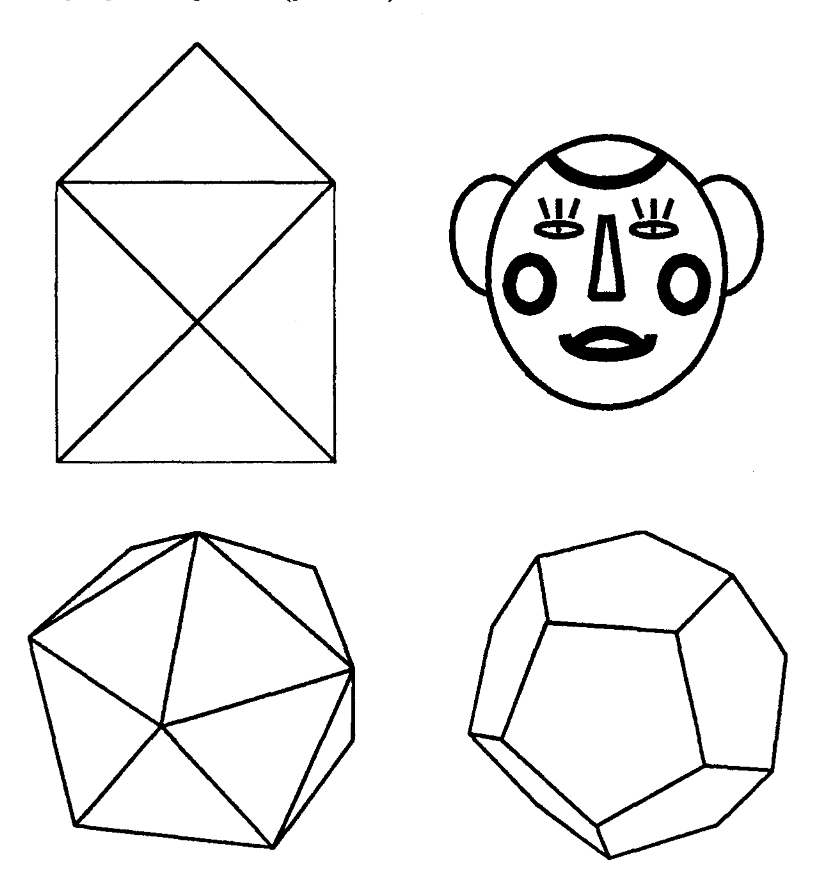


Рис. 3.25

Глава 4
РАБОТА СО СПИСКАМИ

Материал, рассматриваемый в этой главе, занимает центральное место в книге, потому что его изучение позволяет продвинуться вперед к овладению программированием на языке высокого уровня, каким является „Математика", или вернуться назад к использованию ее в качестве калькулятора, но с помощью новых мощных средств работы с выражениями „Математики". Основным объектом этой главы являются списки — один из наиболее фундаментальных способов структурирования данных.

4.1. Порождение списков

Список есть выражение „Математики", имеющее заголовок **List**. Его элементами могут быть любые выражения „Математики". С клавиатуры списки можно вводить с помощью фигурных скобок:

{a, Sin[u + v], List[1, 2, 3]}

$\{a, Sin[u+v]], \{1,2,3\}\}$.

Элементами списка могут быть списки. Наиболее часто встречающаяся конструкция из списка списков — матрицы.

m = {{m11, m12, m13}, {m21, m22, m23},
{m31, m32, m33}} // MatrixForm

$$\begin{matrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{matrix}$$

Функция **MatrixForm** представила список **m** в привычной матричной форме. Возможен пустой список **List**[], или {}. Следующие пять функций порождают списки.

Выражение **Range**[**n**] имеет результатом список $\{1, 2, \ldots, n\}$, выражение **Range**[**m**, **n**], где числа m, n не обязательно целые, — список $\{m, m+1, \ldots, m+ki\}$, причем $m+ki < n < m+ +(k+1)i$:

Range[**2.2**, **4.9**]
{2.2, 3.2, 4.2}

Если у функции **Range** заданы три аргумента, то последний определяет шаг порождения чисел, т.е. разность между последовательными элементами списка:

Range[**2.2**, **4.9**, **0.5**]
{2.2, 2.7, 3.2, 3.7, 4.2, 4.7}

Функция **Table** позволяет генерировать более сложно устроенные списки. Выражение **Table**[**expr**, {**n**}] порождает список из n значений одного и того же выражения *expr*.

Table[**learn**, {**3**}]
{*learn*, *learn*, *learn*}

Здесь необходимо сделать следующее уточнение. Дело в том, что одно и то же исходное выражение *expr* при вычислении может давать различные результаты. Чтобы понять это, рассмотрим функцию **Random**, часто используемую в математическом моделировании. Выражение **Random**[ ] при вычислении имеет результатом однородно распределенную псевдослучайную вещественную величину, заключенную в интервале от 0 до 1. Поэтому вычисление выражения **Table**[**Random**[ ]] может дать, например, следующий результат:

Table[Random[ ], {7}]
{ 0.242873, 0.844294, 0.887767, 0.938646, 0.692790, 0.580143, 0.363467}

В этом случае результат вычисления не состоит из копий одного и того же выражения, как это было в предыдущем примере. Выражение **Random[type, range]** дает псевдослучайные числа типов Integer, Real или Complex, заключенные в интервале *range*.

Table[Random[Integer, {0, 9}], {17}]
{0,0,8,9,9,5,7,3,5,2,6,1,9,3,9,7,4}

В случае типа Real третий аргумент функции **Random** может задавать число цифр, используемых для представления вещественного числа.

Вернемся к функции **Table**. Выражение **Table[expr, {i, n}]** при вычислении приводит к списку из n значений выражения *expr*, отвечающих параметру i, изменяющемуся от 1 до n.

Table[N[Log[i]], {i, 10}]
{0, 0.693147, 1.09861, 1.38629, 1.60944, 1.79176, 1.94591, 2.07944, 2.19722, 2.30259}

Второй аргумент функции **Table** называется *итератор*. Если итератор имеет вид $\{i, m, n\}$, то *expr* вычисляется начиная с $i = m$, если же итератор равен $\{i, m, n, di\}$, то di есть шаг по переменной i. В случае, когда указано несколько итераторов: **Table[expr, {i, imin, imax}, {j, jmin, jmax}, ...]**, порождаются вложенные списки:

Table[i + j, {i, 1, 2}, {j, 1, 3}]
{{2,3,4}, {3,4,5}}

Верхние границы второго и последующих итераторов могут зависеть от переменных предшествующих итераторов. В частности, это позволяет получать треугольные таблицы.

```
Table[x^(i+j),{i,1,3},{j,1,i}] // TableForm
```

x^2
$x^3 \quad x^4$
$x^4 \quad x^5 \quad x^6$

Функция **Array[a, n]** порождает список длины n с элементами $a[i]$ для $i = 1, 2, \ldots, n$. Итератор, заданный в виде $\{n_1, n_2, \ldots\}$, приводит к вложенному списку с элементами $a[i_1, i_2, \ldots]$:

```
Array[a,{3,2}]
{{a[1,1],a[1,2]},{a[2,1],a[2,2]},{a[3,1],a[3,2]}}
```

Функция **Array[a, iterators, origin]** дает список, в котором индексы итераторов изменяются со значения *origin*, по умолчанию равному единице. Если же аргументы функции **Array** заданы в виде **Array[a, iterators, origin, h]**, то получается выражение „Математики“, в котором заголовок List всюду заменен на заголовок h.

```
Array[b,4,2,Plus]
b[2] + b[3] + b[4] + b[5]
```

Выражение **DiagonalMatrix[list]** порождает список, отвечающий двумерной диагональной матрице с элементами из списка *list* на главной диагонали, в то время как функция **IdentityMatrix[n]** дает единичную $n \times n$ матрицу.

4.2. Преобразования списков

К числу простейших преобразований, которые можно проделать над списками, относятся добавление к списку и удаление элементов из списка. С помощью функции **Append** элемент *elem* можно поставить на последнее место списка *list*, а с помощью функции **Prepend** — на первое место в списке.

Append[{a, b, c}, elem]
$\{a, b, c, elem\}$

Prepend[{a, b, c}, elem]
$\{elem, a, b, c\}$

Каждый элемент списка однозначно определяется своим номером, положительным, если отсчет ведется слева, или отрицательным, если отсчет ведется справа. Функция **Insert** позволяет вставить *elem* в список, поместив его на k-е место. Остальные элементы сохраняются в списке.

{Insert[{a, b, c, d}, elem, 2], Insert[{a, b, c, d}, elem, −2]}
$\{\{a, elem, b, c, d\}, \{a, b, c, elem, d\}\}$

Как мы уже отмечали, элементами списка могут быть любые выражения „Математики", в том числе списки. Новый элемент *elem* можно вставить в любой внутренний список в списке *list* или внутрь любого элемента списка, если этот элемент не является атомарным выражением.

Insert[{{a, b, c}, {1, 3}}, 2, {2, 2}]
$\{\{a, b, c\}, \{1, 2, 3\}\}$

Указание позиции в виде $\{2, 2\}$ означает, что новый элемент нужно вставить во второй элемент списка на второе место. Один и тот же элемент можно вставить на несколько позиций, если воспользоваться функцией **Insert** в следующем виде:

Insert[{{a, b, c}, {1, 3}}, 2, {{1, 2}, {2, 1}}]
$\{\{a, 2, b, c\}, \{2, 1, 3\}\}$

Функция **ReplacePart** действует аналогично функции **Insert** с той разницей, что новый элемент замещает элемент списка *list*, стоящий на k-м месте.

ReplacePart[{1, 3}, 2, 2]
$\{1, 2\}$

Удалить элемент списка без его замещения новым элементом можно с помощью функции **Delete**. А именно **Delete[list,k]** приводит к удалению из списка элемента, стоящего на k-м месте. Как и у функций **Insert** и **ReplacePart**, задание позиции с помощью списка $\{k1, k2, \ldots\}$ приводит к удалению элемента, находящегося внутри $k1$-го элемента, в нем — внутри $k2$-го элемента и т.д.

Delete[{a,{1,2,3},b,c},{2,3}]
$\{a, \{1, 2\}, b, c\}$

Для того чтобы удалить несколько элементов, достаточно задать их позиции, заключенные в общие фигурные скобки.

Delete[{a,{1,2,3},b,c},{{2},{3}}]
$\{a, c\}$

Вычисление выражения **Drop[list,k]** приводит к списку с удаленными k первыми элементами (при положительном k) или последними элементами (при отрицательном k). Если k заключить в фигурные скобки, то **Drop** подобно **Delete** удалит только k-й элемент списка, если же вместо k указать список $\{m, n\}$, то будут удалены элементы с m-го по n-й.

Drop[{x^2,y^2,z^2,u^2},-2]
$\{x^2, y^2, u^2\}$

Применение функции **Rest** к списку приводит к новому списку, в котором отсутствует первый элемент прежнего:

Rest[{1,2,3}]
$\{2, 3\}$

Наиболее общей функцией, применяемой для извлечения элементов из списков, является функция **Part**. Выражение **Part[list,k]** дает элемент, стоящий на k-м месте списка. Если нужно извлечь элемент из внутренних списков или из неатомарных элементов списка, то вместо числа k указывается последовательность $k1$, $k2$, ...

Part[{a, Sin[x + y]}, 2, 1, 2]
y

Роль функции **Drop** в паре {**Delete**, **Drop**} по отношению к функции **Part** играет функция **Take**.

{Part[{a, b, c}, 2], Take[{a, b, c}, 2]}
{*b*, {*a*, *b*}}

Функция **First** извлекает первый, а функция **Last** — последний элемент из списка.

Действие описанных только что функций основано на позиции элементов списка. Функция **Select** опирается на свойство элементов удовлетворять заданному критерию. Этот критерий представляет собой предикат, являющийся вторым аргументом рассматриваемой функции.

Select[{a, −3, Cos[x], 37, {u, v}, 15}, NumberQ]
{−3, 37, 15}

Если третьим аргументом у функции **Select** задано натуральное число n, то в получаемый список войдут только первые n элементов, удовлетворяющие критерию.

Select[{a, −3, Cos[x], 37, {u, v}, 15}, NumberQ, 2]
{−3, 37}

Обратить порядок элементов в списке можно с помощью функции **Reverse**, в то время как функция **Sort** располагает элементы в соответствии с принятым в „Математике" порядком. Вкратце этот порядок можно охарактеризовать, сказав, что числа в нем располагаются первыми и по величине, а буквы алфавита — в алфавитном порядке.

Reverse[{c, 3, a}]
{*a*, 3, *c*}

Sort[{c, 3, a}]
{3, *a*, *c*}

Функции **RotateLeft** и **RotateRight** производят циклическое перемещение элементов списка влево или вправо на указанное в качестве второго аргумента этих функций число элементов.

RotateLeft[{a, b, c, d}, 2]
$\{c, d, a, b\}$

Узнать, сколько раз некоторый элемент *elem* встречается в списке, можно с помощью функции **Count**.

Count[{a, b, a}, a]
2

Обратимся теперь к функциям, которые рассматривают список скорее как множество, а не как упорядоченную структуру. Вообще говоря, в списке могут содержаться совпадающие элементы. Применение к нему функции **Union** приводит к удалению повторений элементов и к сортировке оставшихся.

Solve[x^5 – 7x^4 + 19x^3 – 25x^2 + 16x – 4, x]
$\{\{x \to 1\}, \{x \to 1\}, \{x \to 1\}, \{x \to 2\}, \{x \to 2\}\}$

Union[Solve[x^5 – 7x^4 + 19x^3 – 25x^2 + 16x – 4, x]]
$\{\{x \to 1\}, \{x \to 2\},\}$

Если в качестве аргументов функции **Union** указать несколько списков, то результатом будет их теоретико-множественное объединение:

Union[{a, b, c, a}, {b, 3}]
$\{3, a, b, c\}$

Сказанное относительно повторяющихся элементов и сортировке справедливо и по отношению к функциям **Intersection** и **Complement**, порождающим пересечение и дополнение списков. Выражение **Complement[list, list1, list2, ...]** порождает список элементов списка *list*, не содержащихся ни в одном из списков *list*1, *list*2 и т.д.

Complement[{a, b, d, e, c}, {a, e}, {e, b}]

{*c*, *d*}

В конце данного раздела рассмотрим функции, осуществляющие структурные изменения в списках. Функция **Partition** в выражении **Partition[list, n]** разбивает список на неперекрывающиеся части длины n, начиная с первого элемента списка. Если число элементов в списке не делится нацело на n, то последние k $(k < n)$ элементов удаляются из списка.

Partition[{1, 2, 3, 4, 5}, 2]

{{1,2}, {3,4}}

Эта же функция в выражении **Partition[list, n, d]** приводит к разбиению на части длины n с отступом d, т.е. первый элемент второго подсписка имел номер $d+1$ в исходном списке. Таким образом, подсписки перекрываются при $d < n$

Partition[{1, 2, 3, 4, 5, 6, 7, 8, 9}, 3, 2]

{{1,2,3}, {3,4,5}, {5,6,7}, {7,8,9}}

Partition[{1, 2, 3, 4, 5, 6, 7, 8, 9}, 3, 4]

{{1,2,3}, {5,6,7}}

Если функция **Partition** приводит к появлению дополнительного уровня в списке, то функция **Flatten** уменьшает их число. А именно **Flatten** в выражении **Flatten[list]** приводит к одноуровневому списку:

Flatten[{1, {2, {3, 4, {5, 6}}}, {7, 8}}]

{1,2,3,4,5,6,7,8}

а в выражении **Flatten[list, n]** убирает заголовки List только до уровня **n** включительно.

Flatten[{1, {2, {3, 4, {5, 6}}}, {7, 8}}, 2]

{1,2,3,4, {5,6}, 7,8}

Модификацией функции **Flatten** является функция **FlattenAt**, которая применяет **Flatten** к частям списка.

FlattenAt[{1,{2,{3,4,{5,6}}},{7,8}},3]
{1,{2,{3,4,{5,6}}},7,8}

Очень интересной функцией является **Transpose**. Если она применяется к списку, представляющему собой прямоугольную матрицу, то результатом будет транспонированная матрица. Пусть **m = {{a,b,c},{1,2,3}}**, тогда

m // MatrixForm
Transpose[m] // MatrixForm

a	*b*	*c*
1	2	3

a	1
b	2
c	3

Более содержательную структурную перестройку вложенных списков можно осуществить в случае, когда список содержит более одного уровня и представляет не обычную двухмерную матрицу, а матрицу более высокой размерности. Рассмотрим в качестве примера трехмерную матрицу

m = {{{a,b},{c,d},{e,f}},
{{1,2},{3,4},{5,6}},
{{aa,bb},{cc,dd},{ee,ff}},
{{11,12},{13,14},{15,16}}}

Список *m* содержит четыре элемента, каждый из которых является списком и состоит из трех списков по два элемента в каждом, которые являются символами или числами. Позиции этих символов и чисел можно охарактеризовать с помощью трех чисел — координат: первое указывает, в какой из строк находится рассматриваемый символ или число, второе — в каком

из списков строки оно содержится, третье — его положение в самом внутреннем списке. Таким образом, каждый символ или число можно рассматривать как матричный элемент m_{ijk} Транспонирование матрицы m означает перестановку индексов. Всего возможны шесть транспозиций. Транспозиция $\{3,1,2\}$ означает, что в новой матрице первый индекс есть второй индекс исходной матрицы, второй индекс есть прежний третий, а третий индекс — прежний первый. Приведем в качестве примера две транспозиции матрицы, или списка, m.

Transpose[m, {2, 1, 3}]

$\Big\{\{\{a,b\},\{1,2\},\{aa,bb\},\{11,12\}\},$
$\{\{c,d\},\{3,4\},\{cc,dd\},\{13,14\}\},$
$\{\{e,f\},\{5,6\},\{ee,ff\},\{15,16\}\}\Big\}$

Transpose[m, {3, 1, 2}]

$\Big\{\{\{a,1,aa,11\},\{b,2,bb,12\}\},$
$\{\{c,3,cc,13\},\{d,4,dd,14\}\},$
$\{\{e,5,ee,15\},\{f,6,ff,16\}\}\Big\}$

Последней функцией, которую мы упомянем в этом разделе, является функция **Join**, осуществляющая сцепление (конкатенацию) списков.

Join[{a, b, c, }, {1, 2}, {x, y}]

$\{a,b,c,1,2,x,y\}$

4.3. Работа с векторами и матрицами

Векторы в „Математике“ трактуются как линейные, т.е. одноуровневые, списки: $v=\{v1,v2,v3\}$; матрицы как двухуровневые: $m=\{\{m11,m12,m13\},\{m21,m22,m23\}\}$, хотя компоненты векторов и матриц могут быть произвольными выражениями. Ввиду важной роли этих объектов в теоретических и прикладных вопросах в „Математике“ определены несколько функций, предназначенных для работы с векторами и матрицами.

Функция **Det** вычисляет детерминант квадратной матрицы.

m1 = {{1,x1,x1^2},{1,x2,x2^2},{1,x3,x3^2}};
Det[m1] // Simplify
$(-x1+x2)(-x1+x3)(-x2+x3)$

Выражение **Minors[matrix,k]** имеет результатом список миноров k-го порядка матрицы *matrix*.

Minors[m1,2]
$\{\{-x1+x2, -x1^2+x2^2, -(x1^2x2)+x1x2^2\},$
$\{-x1+x3, -x1^2+x3^2, -(x1^2x3)+x1x3^2\},$
$\{-x2+x3, -x2^2+x3^2, -(x2^2x3)+x2x3^2\}\}$

Функция **Inverse** вычисляет обратную матрицу для невырожденных квадратных матриц.

inversm1 = Inverse[m1] // Simplify

$$\left\{\left\{\frac{x2\,x3}{(x1-x2)(x1-x3)}, \frac{x1\,x3}{(-x1+x2)(x2-x3)}, \frac{x1\,x2}{(x1-x3)(x2-x3)}\right\},\right.$$
$$\left\{\frac{x2+x3}{(-x1+x2)(x1-x3)}, \frac{x1+x3}{(-x1+x2)(-x2+x3)},\right.$$
$$\left.\frac{x1+x2}{(x2-x3)(-x1+x3)}\right\}, \left\{\frac{1}{(-x1+x2)(-x1+x3)},\right.$$
$$\left.\left.\frac{1}{(-x1+x2)(x2-x3)}, \frac{1}{(x1-x3)(x2-x3)}\right\}\right\}$$

Скалярное произведение векторов, произведение вектора и матрицы, а также произведение матриц вычисляются с помощью функции **Dot**. Входной формат для этой функции есть **expr1.expr2**. Функция **Dot** определена для двух, трех и большего числа аргументов. В самой общей форме результатом применения этой функции к матрицам (тензорам) $M_{i_1 i_2 \ldots i_n}$ и $N_{j_1 j_2 \ldots j_m}$ является матрица $\sum_k M_{i_1 i_2 \ldots k} N_{k j_2, \ldots j_m}$.

Приведем несколько примеров употребления функции **Dot**.

v1 = {1,0,0}; v2 = {a,b,c};
v1.v2
a

v1.m1
$\{1, x1, x1^2\}$

m1.v1
$\{1, 1, 1\}$

m1.inversm1
$\{\{1,0,0\},\{0,1,0\},\{0,0,1\}\}$

С помощью функции **MatrixPower** можно вычислять целые степени матрицы.

MatrixPower[m1,2]
$\{\{1 + x1 + x1^2, x1 + x1x2 + x1^2x3, x1^2 + x1x2^2 + x1^2x3^2\},$
$\{1 + x2 + x2^2, x1 + x2^2 + x2^2x3, x1^2 + x2^3 + x2^2x3^2\},$
$\{1 + x3 + x3^2, x1 + x2x3 + x3^3, x1^2 + x2^2x3 + x3^4\}\}$

Собственные числа и собственные векторы матриц можно найти, используя функции **Eigenvalues** и **Eigenvectors**, их совокупность — используя **Eigensystem**, а функция **CharacteristicPolynomial** дает характеристический полином матрицы:

m2 = {{a,1,0},{-1,a,1},{0,-1,a}};
Eigenvalues[m2]
$\{a, -ISqrt[2] + a, ISqrt[2] + a\}$
Eigenvectors[m2]
$\{\{1,0,1\},\{-1, ISqrt[2], 1\},\{-1, -ISqrt[2], 1\}\}$
Eigensystem[m2]
$\{\{a, -ISqrt[2] + a, ISqrt[2] + a\},$

$\{\{1,0,1\},\{-1, ISqrt[2], 1\},\{-1, -ISqrt[2], 1\}\}\}$
CharacteristicPolynomial[m2,x]
$2a + a^3 - 2x - 3a^2x + 3ax^2 - x^3$

4.4. Выражения „Математики“

Как мы уже отмечали в разделе 1.2, основу работы компьютерной алгебры „Математика“ составляет вычисление выражений. Общий вид последних есть

h[e1,e2,...,en],

где h — заголовок, а e_1, e_2, ..., e_n — элементы, или непосредственные подвыражения, или части выражения. И заголовок, и элементы в свою очередь являются выражениями. Исходным материалом для построения выражений служат атомарные выражения: символы (последовательность латинских букв, цифр и знака $, не начинающаяся с цифры), числа (целые, рациональные, вещественные, комплексные) и строки. Списки как выражения „Математики“ не являются какими-либо особенными объектами компьютерной алгебры, но их описание выделено в отдельную главу по двум причинам. Во-первых, списки сравнительно простой для восприятия и понимания объект, удобный для изучения некоторых общих приемов работы с выражениями. Во-вторых, обработка списков есть фундамент программирования, так как в „Математике“ они служат для представления разнообразных объектов, традиционно относимых в других языках программирования к векторам, матрицам и другим типам данных. В этом разделе мы дадим описание некоторых функций, применяемых для преобразования выражений.

Полной формой выражения называется запись этого выражения, в которой и заголовок, и элементы представлены в свою очередь в виде выражений „Математики“. Допустим, что с клавиатуры введено выражение

```
{a,b,D[f[x,y],{x,2},y],Integrate[(z+1)/(z+2),z]}
```

Его полная форма есть

```
List[a,b,D[f[x,y],{x,2},y],Integrate[Times[Plus[z,1],
Power[Plus[z,2],-1]],z]]
```

Полную форму *вычисленного* выражения можно получить с помощью функции **FullForm**.

{a, b, D[f[x, y], {x, 2}, y],
Integrate[(z + 1)/(z + 2), z]} // FullForm
{a, b, Derivative[2, 1][f][x, y],
Plus[z, Times[-1, Log[Plus[2, z]]]]}

Мы видим, что прежде чем вычислить полную форму, „Математика“ вычислила интеграл. Для получения полной формы невычисленного выражения *expr* нужно применить к нему функции **Hold** или, что дает тот же эффект, **HoldForm**, которые блокируют вычисление *expr*.

FullForm[HoldForm[{a, b, D[f[x, y], {x, 2}, y],
Integrate[(z + 1)/(z + 2), z]}]]
HoldForm[List[a, b, D[f[x, y], List[x, 2], y],
Integrate[Times[Plus[z, 1], Power[Plus[z, 2], -1]], z]]]

Если вычисленное выражение *expr* атомарно, то предикат **AtomQ**, примененный к нему, имеет результатом *True*, в противном случае — *False*. Заголовок выражения можно получить после вычисления выражения **Head[expr]**. Число n элементов является значением выражения **Length[expr]**. Элемент e_k извлекается из выражения с помощью функции **Part** в виде **Part[expr, k]** или в виде **expr[[k]]**. С помощью этой функции можно также получить заголовок выражения. Для того достаточно вычислить выражение **Part[expr, 0]**, или **expr[[0]]**. Поскольку каждый элемент e_k выражения общего вида является самостоятельным выражением, то можно извлекать части из него и из его элементов и т.д. Для этого вместо одного числа — номера k элемента e_k нужно в функции **Part** указать последовательность номеров соответствующих элементов.

expr1 = {a, b, D[f[x, y], {x, 2}, y], Integrate[(z + 1)/(z + 2), z]}

{expr1[[2]], expr1[[3, 0]], expr1[[3, 0, 1]], expr1[[3, 0, 0, 1]]}

$\{b, f^{(2,1)}, f, 2\}$

Рассматриваемая последовательность номеров элементов называется *спецификацией элемента.* Спецификация элемента является значением функции **Position**.

Position[expr1, z]

{{4, 1}, {4, 2, 2, 1, 2}}

Введем понятие уровней в выражении и глубины выражения. Элементы e_1, e_2, ..., e_n находятся на первом уровне выражения. Их элементы находятся на втором уровне и т.д. Количество чисел в спецификации элемента позволяет узнать, на каком уровне он находится. *Глубина* выражения численно равна максимальному номеру уровня в выражении плюс единица и является значением функции **Depth**. Функция **Level** имеет значением список всех частей выражения, находящихся на указываемом в качестве ее второго аргумента уровне.

Level[expr1, {2}]

$\{x, y, z, -Log[2 + z]\}$

Если второй аргумент задан без скобок, то значением будет список всех частей выражения, находящихся на уровнях от первого до указанного.

Level[expr1, 2]

$\{a, b, x, y, f^{(2,1)}[x, y], z, -Log[2 + z], z - Log[2 + z]\}$

Если в качестве второго аргумента дана пара чисел $\{m, n\}$, заключенная в фигурные скобки, то будет получен список всех частей на уровнях от m до n включительно.

Level[expr1, {3, 4}]
$\{-1, 2+z, Log[2+z]\}$

Depth[expr1]
6

В случае когда второй аргумент функции **Level** — отрицательное число, заключенное в фигурные скобки, например $\{-k\}$, то значением функции будет список всех частей вычисленного выражения, чья глубина равна k.

Level[expr1, {−2}]
$\{f^{(2,1)}[x,y], 2+z\}$

Если же отрицательное число не заключать в фигурные скобки, то список будет содержать все подвыражения вычисленного выражения, чья глубина по меньшей мере равна k, т.е. k и больше.

Level[expr1, −2]
$\{f^{(2,1)}[x,y], 2+z, Log[2+z], -Log[2+z], z-Log[2+z]\}$

Указание уровней выражения только что рассмотренными способами называется *спецификацией уровня.*

Имеются четыре основных способа интерпретации выражений.

1. Заголовок h выражения можно понимать как название какой-то математической функции, например Sin, а элементы выражения — как аргументы этой функции.

2. Заголовок можно понимать как команду (Factor), а элементы — как ее адресаты.

3. Заголовок есть оператор или операция (Integrate, Times), применяемый к операндам.

4. Заголовок может указывать на тип данных (List, Real).

В конце раздела уместно сделать следующее важное замечание: все функции, рассмотренные в предыдущем разделе в связи со списками, можно применять к произвольным выражениям.

expr2 = h[x, h[x + y]];
Flatten[expr2]
h[x, x + y]

expr3 = ReplacePart[expr2, z, Position[expr2, y]]
h[x, h[x + z]]

Union[expr2, expr3]
h[x, h[x + y], h[x + z]]

В рассмотренных примерах *h* — произвольный заголовок.

4.5. Вычисление функций от списков и их элементов

В этом разделе нам будет удобно принять первую из упомянутых выше интерпретаций выражений „Математики“, когда заголовок выражения понимается как название функции. Значение функции на совокупности аргументов мы будем называть применением функции (точнее, заголовка функции) к аргументам. Таким образом, результатом применения функции *f* к списку *list* является выражение *f*[*list*], входные формы которого могут быть *f*@*list* или *list* // *f*. Если мы попытаемся вычислить значения многих встроенных функций „Математики“ на списках, то столкнемся с неожиданным результатом.

Sqrt[{1, 2, 3}], или **Sqrt@{1, 2, 3}**, или **{1, 2, 3} // Sqrt**
{1, *Sqrt*[2], *Sqrt*[3]}

Вместо математически неопределенного выражения **Sqrt[{1, 2,3}]** мы получили список квадратных корней из элементов списка. Еще пример:

x^{1,2,3}, или **Power[x,{1,2,3}]**
$\{x, x^2, x^3\}$

Эти примеры демонстрируют свойство встроенных функций, которое можно назвать *дистрибутивностью* относительно списков. Если же к списку применить произвольную функцию f, то она, не обладая свойством дистрибутивности, даст ожидаемый результат:

f@{1,2,3}
$f[\{1,2,3\}]$.

Функция **Map** заставляет другие функции вести себя так, как если бы они были дистрибутивны.

matrix = {{m11, m12, m13}, {m21m22, m23}, {m31, m32, m33}}
Map[f, matrix], или **f/@matrix**
$\{f[\{m11, m12, m13\},], f[\{m21, m22, m23\},], f[\{m31, m32, m33\}]]\}$

Если в качестве третьего аргумента у функции **Map** указана спецификация уровня, то заголовок f будет применяться к элементам соответствующих уровней.

Map[f, matrix, {2}]
$\{\{f[m11], f[m12], f[m13]\}, \{f[m21], f[m22], f[m23]\}, \{f[m31], f[m32], f[m33]\}\}$

Модификацией рассматриваемой функции служит функция **MapAt**. Она применяет заголовок f к элементам списка в соответствии с их спецификацией.

MapAt[f, matrix, {{1,1},{2,2},{3,3}}]
{{f[m11], m21, m23}, {m21, f[m22], m23}, {m31, m32, f[m33]}}

Обладает ли встроенная функция свойством дистрибутивности относительно списков, можно узнать, вычислив выражение **Attributes[function]**. Если в списке атрибутов функции *function* присутствует Listable, то она дистрибутивна. Если нет, то в необходимых случаях следует прибегнуть к **Map**.

{Attributes[Sqrt], Attributes[Reverse]}
{{*Listable*, *Protected*}, {*Protected*}}

Из двух функций **Sqrt** и **Reverse** первая обладает свойством дистрибутивности, в чем мы уже убедились раньше, вторая нет. Смысл атрибута Protected будет объяснен ниже.

l = {{a, b}, {c, d}};
{Reverse[l], Map[Reverse, l]}
{{{c, d}, {a, b}}, {{b, a}, {d, c}}}

Еще более неожиданный результат получается, если дистрибутивную функцию применить к нескольким спискам.

{a, b}^{2, 3}, или **Power[{a, b}, {2, 3}]**
$\{a^2, b^3\}$

Другой пример — обычное произведение трех спискoв:

{a, b} {2, 3} {x, y}
$\{2ax, 3by\}$

Примеры показывают, что при применении дистрибутивной функции от нескольких аргумéнтов к спискам одинаковой длины происходит следующее: сначала производится транспозиция списков, т.е. объединяются в списки элементы с одинаковыми номерами из разных списков, потом функция применяется к вновь образованным спискам так , что элементы этих списков становятся аргументами функции. Проиллюстрируем сказанное более детально на примере.

ll = {{a, b}, {2, 3}, {x, y}};
tll = Transpose[ll]
$\{\{a,2,x\},\{b,3,y\}\}$
Map[f, tll]
$\{f[\{a,2,x\}],f[\{b,3,y\}]$

Если бы $a,2,x$ и $b,3,y$ были бы аргументами функции f, то получился бы список $\{f[a,2,x],f[b,3,y]\}$, который для $f = Times$ совпадал бы со списком рассмотренного нами примера. Таким образом, дополнительно происходит замена заголовка List на заголовок f. Такую замену производит встроенная функция **Apply**.

Apply[f, {a, 2, x}], или **f@@{a, 2, x}**
$f[a,2,x]$

Все эти операции в совокупности производит функция **MapThread**.

MapThread[f, {{a, b}, {2, 3}, {x, y}}]
$\{f[a,2,x],f[b,3,y]\}$

В „Математике" имеется другая, в чем-то похожая, но в то же время существенно отличающаяся от **MapThread** функция, а именно **Thread**. Предположим, что имеется вычисленное

„Математикой“ выражение вида **f[list$_1$,list$_2$,...]**. Функция **Thread** сделает с заголовком *f* то же самое, что и функция **MapThread** со „свободным“ заголовком *f*.

Thread[f[{a, b}, {1, 2}]]
{f[a, 1], f[b, 2]}}

Если же выражение **f[list$_1$, list$_2$,...]** при вычислении изменяется, то **MapThread** и **Thread** дают разные результаты.

MapThread[SameQ, {{a, b}, {a, c}}]
{True, False}

Thread[SameQ[{a, b}, {a, c}]]
Thread:: normal:
Normal expression expected at position 1 in Thread[False]
Thread[False]

Мы видим, что при применении **MapThread** функция **SameQ** вычислялась на парах (a, a) и (a, c), а при применении **Thread** сначала было вычислено выражение **SameQ[{a,b},{a,c}]** с результатом False. Кроме того, если **MapThread** в качестве второго аргумента может иметь только список списков, **Thread** „продевает“ заголовок выражения сквозь элементы подвыражений с иными заголовками, чем List.

Thread[f[x == y, a == b], Equal]
f[x, a] == f[y, b]
MapThread[f, {x == y, a == b}]
MapThread:: mptd:
Object x==y at position (2, 1) has only 0 of required 1 dimensions
MapThread[f, {x == y, a == b}]

Мы рассматривали действие функции **Thread** в случаях, когда списки или выражения — ее аргументы были одинаковой длины. Возможно, что один или несколько аргументов являются символами. Тогда этот символ копируется столько раз, какова длина аргументов-списков.

Thread[f[x, {a, b}, {1, 2}]]
$\{f[x, a, 1], f[x, b, 2]\}$

Встроенная функция **Inner** обобщает как ранее рассмотренную функцию **Dot**, так и функцию **MapThread**. Выражение **Inner[f, list₁, list₂, ..., g]** эквивалентно выражению **Apply[g, MapThread[f, {list₁, list₂, ...]]**. В случае $g = Plus$ и $f = Times$ функция **Inner** совпадает с **Dot**. Кроме того, аргументы $list_1$, $list_2$ и т.д. могут иметь одинаковые заголовки, не обязательно совпадающие с List.

Inner[Times, a b, x y, Power]
$(a\,x)^{b\,y}$

Функция **Outer** в выражении **Outer[f, list₁, list₂, ...]** применяет f ко всем элементам прямого (декартова) произведения списков $list_1$, $list_2$ и т.д.

Outer[f, {a, b}, {u, v, w}]
$\{\{f[a, u], f[a, v], f[a, w]\}, \{f[b, u], f[b, v], f[b, w]\}\}$

Более точное описание действия этой функции состоит в следующем. Для матриц (тензоров) $M_{i_1 i_2, \ldots i_p}$ и $N_{j_1 j_2 \ldots j_q}$ результат применения заголовка f к M и N есть тензор порядка $p + q$ с компонентами $f[M_{i_1 i_2, \ldots i_p}, N_{j_1 j_2 \ldots j_q}]$. Декартово произведение списков получается, если в качестве f выбрать заголовок List и применить к полученному результату **Flatten**.

Flatten[Outer[List, {x,y}, {1,2}], 1]
$\{\{x,1\},\{x,2\},\{y,1\},\{y,2\}\}$

Последнее замечание относительно функции **Outer** состоит в том, что заголовки выражений $list_i$ не обязательно должны быть List, хотя и обязаны быть одинаковыми.

Outer[Power, x + y, u + v]
$x^u + x^v + y^u + y^v$

В общем случае заголовка g у второго, третьего и т.д. аргументов этот заголовок будет иметь все вычисленное выражение и его элементы.

Outer[f, g[x,y], g[1,2]]
$g[g[f[x,1],f[x,2]],g[f[y,1],f[y,2]]]$

Упражнения

1. По списку $l = \{a,b,c,d,e,f,g\}$ *постройте* новый список $l3$, содержащий каждый третий элемент l. Из l *удалите* каждый третий элемент.
2. Из списка $l = \{a,1,b,0,c,0,0,d\}$ *исключите* все нулевые элементы.
3. *Найдите* сумму отрицательных чисел в списке $l = \{a,2,-1,b,c,-3\}$.
4. *Найдите* среднее арифметическое квадратов элементов списка $l = \{-3,4,-7,0,11\}$.
5. В списке $l = \{3,-2,-5,7\}$ *найдите* элемент с минимальным абсолютным значением.
6. *Найдите* позицию первого отрицательного элемента в числовом списке l.
7. *Преобразуйте* числовой список l в True, если в нем есть простые числа, или в False в противном случае.
8. Даны матрица m и столбец l. Как вставить l k-м столбцом в матрицу?
9. Как с помощью функции **Table** получить список $\{\{0\},\{0,1\},\{0,1,4\},\{0,1,4,9\},\{0,1,4,9,16\}\}$?

10. (Случайное блуждание по двухмерной решетке). *Получите* список m из двадцати пар, порождаемых по следующему закону. Элементами первой пары являются случайно выбранные числа из множества $\{-1, 1\}$. Элементы второй пары получаются прибавлением к первой аналогичной случайной пары и т.д. После того как список m будет получен, *вычислите* выражение

    Show[Graphics[{Line[m]}]]

11. Пусть $f(x,y) = x + y^3$, $g(x,y) = x^2 - y^2 - y$. *Подсчитайте* якобиан этих функций в точке $(0,0)$.

12. Дана функция $f(x,y,z) = xy + z$. *Подсчитайте* градиент этой функции, т.е. *получите* список ее первых частных производных, с помощью функций **Outer** и **D**.

13. Дано векторное поле **v** в трехмерном пространстве:

    $$\mathbf{v} = \{f(x,y,z), g(x,y,z), h(x,y,z)\}.$$

    Получите выражение для дивергенции этого векторного поля с помощью функций **Inner** и **D**.

14. Даны список переменных $l = \{x, y, z\}$ и список их значений $m = \{1, 2, 3\}$. *Получите* список подстановок $r = \{x \to 1, y \to 2, z \to 3\}$.

15. Даны матрица $m = \{\{1,2\},\{3,4\}\}$, список неизвестных $l = \{x, y\}$ и список правых частей линейной системы уравнений $r = \{0, 1\}$. *Получите* линейную систему алгебраических уравнений с данной матрицей коэффициентов и списком правых частей.

ЧАСТЬ II

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМПЬЮТЕРНОЙ АЛГЕБРЫ „МАТЕМАТИКА“

„Математика“ задумана и выполнена с целью максимально упростить для пользователя компьютерную реализацию математических алгоритмов и методов. „Математика“, в сущности, является языком программирования высокого уровня, поэтому написание программ есть наиболее адекватная и эффективная форма взаимодействия с ней. Рассматриваемая версия системы позволяет программировать в трех стилях: 1) функциональном, 2) стиле, основанном на создании правил преобразований, 3) процедурном, хотя допускает и смешанный характер программирования. В этом разделе будут последовательно рассмотрены перечисленные стили программирования.

Глава 5
ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

5.1. Функции, определяемые пользователем

В „Математике" около тысячи встроенных функций. Однако после определенного числа сеансов работы с ней пользователь начинает замечать, что при проведении расчетов в интересующей его области определенные комбинации встроенных функций повторяются, и неплохо было бы поэтому последовательное применение нескольких встроенных функций заменить применением одной. Это простое замечание и лежит, в сущности, в основе функционального программирования, в котором функции применяются к аргументам, затем другие функции применяются к функциям от аргументов, затем новые функции применяются к функциям от функций от аргументов и т.д. Помимо встроенных функций пользователю могут понадобиться совсем простые функции, типа возведения аргумента в куб, отсутствующие среди встроенных функций.

Самое простое, что приходит в голову при попытке определить функцию $f(x) = x^3$, — это присвоить выражению **cube[x]** значение **x^3**:

```
cube[x] = x^3
```

x^3

Прием, когда вместо невыразительного заголовка f для функций используется заголовок, несущий информацию о ее назначении, является довольно обычным, поэтому мы и выбрали за-

головок *cube* для рассматриваемой функции. Если попытаться вычислить значения функции *cube* для аргумента 2 или аргумента y, то ожидаемых результатов 8 или y^3 не получится.

{cube[2], cube[y], cube[x]}
{*cube*[2], *cube*[y], x^3}

Вычисление произошло только для аргумента x, и все дело в том, что пользователь склонен рассматривать x как переменную, пробегающую по области определения функции, в то время как для „Математики" x — это фиксированный символ. Выражения *cube*[x] и *cube*[y] отличны друг от друга, поэтому если вместо первого выражения всюду в дальнейшем будет подставляться x^3, то второе выражение будет оставаться неизменным, в чем мы, собственно, и убедились.

Существует способ объяснить „Математике", что в квадратных скобках после заголовка **cube** стоит переменная. Для этого нужно вместе с символом x употребить выражение **Blank**[ ], которое во входном формате совпадает с подчеркиванием _:

cube[x_] := x^3

В этом определении использовано отложенное присвоение **SetDelayed**, поэтому выходной строки нет. Почему использовано отложенное присвоение, мы подробно объясним позже, а пока сделаем простое замечание: опыт показывает, что с отложенным присвоением возникает меньше недоразумений при применении функций в дальнейшем. Теперь наше определение превосходно работает:

{cube[2], cube[y], cube[x]}
$\{8, y^3, x^3\}$

Есть еще два способа указать „Математике", что некоторое выражение следует рассматривать как функцию определенных

переменных. С ними мы уже встречались в предыдущих главах. Вспомним, что типичные операторы математического анализа: дифференцирование и интегрирование — можно было применять к произвольным выражениям *expr*. По каким переменным *expr* рассматривается как функция, указывалось заданием второго, третьего и т.д. аргументов:

```
D[a x^2 + b y, a]
```
x^2

Аналогично обстояло дело и с графическими функциями **Plot**, **ParametricPlot** и т.п. Второй способ: с помощью функции **ReplaceAll** мы могли вычислять значение выражения для отдельных символьных, численных и других значений входящего в выражение символа, тем самым проделывая типичные для функций процедуры подстановки конкретных значений аргумента или суперпозиции функций.

```
a x^2 + b y /. x -> 2
```
$4a + by$

Теперь мы можем сделать из выражения $expr = ax^2 + by$ традиционную функцию двух аргументов x и y, рассматривая символы a и b как параметры:

```
poly[x_, y_] := a x^2 + b y
{D[poly[x, y], y], poly[2, y]}
```
$\{b,\ 4a + by\}$

В качестве еще одного примера определим очень простую функцию для работы над списками с численными элементами, а именно функцию, вычисляющую среднее арифметическое списка:

```
average[x_] := Apply[Plus, x]/Length[x]
average[{13, 7, 16}]
```
12

Выше были определены три новые функции. С этого момента с их заголовками можно обращаться в точности так же, как и с заголовками встроенных функций.

Map[cube,{a,b,3}]
$\{a^3, b^3, 27\}$

MapThread[poly,{{x,y,z},{u,v,w}}]
$\{bu + ax^2, bv + ay^2, bw + az^2\}$

5.2. Чистые и анонимные функции

В предыдущем разделе определяемые пользователем функции снабжались заголовками, и в этом смысле их можно назвать именованными функциями. Последние имеет смысл определять, если эти функции будут использованы в дальнейшем в программе или в течение сеанса интерактивной работы. В случае, когда новая функция возникает по ходу дела и нет оснований ожидать, что она вновь потребуется, полезно работать с так называемыми *чистыми* функциями, которые используются только в момент их создания. Для определения чистых функций служит встроенная функция **Function**. Вот как можно другим способом определить функцию **cube**: это будет **Function[{x},x^3]**. В этом определении x — локальная переменная, которую вполне можно заменить на y, z или любой другой символ, поэтому значения глобального символа x не влияют на определяемую чистую функцию. Применение чистой функции к аргументу происходит стандартным образом:

{Function[{x},x^3][2],Function[{x},x^3][y],
Function[{x},x^3][x]}
$\{8, y^3, x^3\}$

Map[Function[{x},x^3]],{a,b}]
$\{a^3, b^3\}$

Чистую функцию легко превратить в именованную: для этого достаточно сделать присвоение символу, который в дальнейшем будет служить ее именем.

newcube = Function[{x},x^3]

Function[{x}, x^3]

newcube[y]

y^3

Подводя итоги, отметим, что **Function** позволяет по любому выражению *expr* определить чистую функцию **Function[{x, y,...},expr]** или именованную функцию **f = Function[{x,y,...}, expr]** переменных x, y, ..., а также из именованной функции сделать чистую **F[{x,y,...},f[x,y,...]]**. В силу того что символы в скобках в **Function** локальны, выражение *expr* должно быть впечатано в момент написания чистой функции. К функциям, определяемым с помощью отложенных присвоений, это замечание не относится.

Существует более компактная и удобная форма представления чистой функции, которую иногда называют *анонимной* функцией. В анонимных функциях вместо связанных переменных используются специальные выражения „Математики" с заголовком **Slot**, имеющие вид #, #1 и т.д. При этом для функций одной переменной используется #, а в функциях нескольких переменных первый аргумент обозначается #1, второй #2 и т.д. Кроме того, в конце анонимной функции ставится знак &. Функция **newcube**, переписанная в форме анонимной функции, выглядит как **#^3&**. Анонимные функции позволяют давать очень элегантные определения функций. Рассмотрим в качестве примера функцию **jacob**, которая по заданным n функциям f_1, f_2, ..., f_n от n переменных x_1, x_2, ..., x_n вычисляет их якобиан: $||\partial f_i/\partial x_j||$.

jacob[f_,x_] := Outer[D[#1,#2]&,f,x]

В этом определении под f понимается список функций, а под x — список независимых переменных, являющихся аргументами функций из списка f. В анонимной функции **D[#1,#2]&** при ее вычислении на место первого аргумента #1 подставляется функция, на место второго аргумента #2 — независимая переменная в соответствии с определением функции **Outer**.

jacob[{f[x,y],g[x,y]},{x,y}] // MatrixForm

$f^{(1,0)}[x,y]$ $f^{(0,1)}[x,y]$
$g^{(1,0)}[x,y]$ $g^{(0,1)}[x,y]$

Анонимные функции очень удобны для работы с функциями **Sort** и **Select**. Функция **Sort** имеет второй, необязательный аргумент, задающий критерий сортировки. Если этот аргумент не указан, то используется встроенный порядок. Второй аргумент обязан быть двухместным предикатом, и сортировка производится так, чтобы он принимал значение True на любой паре последовательных элементов списка. Рассмотрим в качестве примера список

l = {5,3,7,1,8,9,2,4};

и упорядочим его так, чтобы квадрат предшествующего элемента был строго больше последующего. Соответствующий предикат задается анонимной функцией **#1^2 > #2&**, поэтому искомая сортировка выполняется с помощью выражения

Sort[l,#1^2 > #2&]

{5,3,7,8,9,4,2,1}

Теперь поставим задачу: выбрать из списка l те элементы, квадратные корни которых меньше 2.5.

Select[l,N[Sqrt[#]] < 2.5&]

{5,3,1,4,2}

5.3. Суперпозиция функций

Важнейшей компонентой функционального программирования является последовательное применение, или суперпозиция функций. Имеются две полезные встроенные функции **Nest** и **Fold**, которые автоматизируют написание суперпозиций одной и той же функции. Предположим, что нужно последовательно n раз применить функцию **function** к некоторому аргументу, затем к результату применения функции к аргументу и т.д. Такая ситуация возникает, например, в итерационных методах. Выражение **Nest[function, x, n]** при его вычислении выполнит эту последовательность действий.

Nest[cube, a, 3]

a^{27}

Nest[D[#, x]&, (x^2 + ax + b)E^x, 25]

$E^x(600 + 25a + b + 50x + ax + x^2)$

В первом примере вычислено выражение *cube[cube[cube[a]]]*, во втором найдена 25-я производная функции $(x^2 + ax + b)e^x$.

В методе касательных Ньютона для нахождения нуля функции $f(x)$, находящегося вблизи от точки a, n-е приближение для искомого нуля находится по формуле

$$x_0 = a,$$
$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}.$$

Эти итерации легко реализуются с помощью функции

newtoniter[f_, x0_, n_] := Nest[(# − f[#]/f'[#])&, N[x0], n],

где x_0 — начальная точка, а функция **N** применена к x_0 для того, чтобы проводить численные, а не символьные вычисления. Вычислим несколько приближений к корню полинома $x^3 - 2x^2 + 5$, расположенному вблизи -1.

```
newtoniter[Function[{x},x^3 - 2x^2 + 5],-1,2]
-1.243
newtoniter[Function[{x},x^3 - 2x^2 + 5],-1,3]
-1.2419
```

Чтобы оценить сходимость и не повторять вручную вычислений, отличающихся только количеством итераций, поступим следующим образом:

```
newtoniter[Function[{x},x^3 - 2x^2 + 5],-1,
#]&/@Range[5]
{-1.28571,-1.243,-1.2419,-1.2419,-1.2419}
```

Здесь по функции **newtoniter** определена анонимная функция аргумента, который указывает число итераций. Она применяется ко всем элементам списка *Range*[5] = {1,2,3,4,5}. Альтернативный способ основан на применении функции **Table** в виде

```
Table[newtoniter[Function[{x},x^3 - 2x^2 + 5],-1,i],
{i,5}].
```

В функциональном программировании отдают предпочтение первому способу. Мы видим, что после третьей итерации происходит стабилизация пяти знаков после запятой, поэтому с этой точностью приближенное значение корня рассматриваемого полинома равно −1.24190.

Функция **FixedPoint** осуществляет итерации до тех пор, пока с машинной точностью результат не перестанет изменяться. Перепишем с ее помощью функцию **newtoniter**:

```
newton[f_,x0_] := FixedPoint[(# - f[#]/f'[#])&,N[x0]]
newton[Function[{x},x^3 - 2x^2 + 5],-1]
-1.2419
```

Еще одна разновидность **NestList** функции **Nest** имеет результатом список, первым элементом которого является аргумент, вторым — результат применения итерируемой функции к аргументу и т.д.

```
newtonlist[f_,x0_,n_] := NestList[(# - f[#]/f'[#])&,
N[x0],n]
newtonlist[Function[{x},x^3 - 2x^2 + 5],-1,5]
{-1.,-1.28571,-1.243,-1.2419,-1.2419,-1.2419}
```

Пара функций **Fold** и **FoldList** является аналогом **Nest** и **NestList** применительно к функциям от двух аргументов. Выражение **FoldList[f,x,{a,b,...}]** порождает список **{x,f[x,a],f[f[x,a],b],...}**, а функция **Fold** от тех же аргументов имеет значением последний элемент этого списка. Рассмотрим два примера. В первом определим функцию **fact[n]**, вычисляющую $n!$:

```
fact[n_] := Fold[Times,1,Range[n]]
fact[5]
120
```

В качестве второго примера рассмотрим задачу: получить список производных функции $x/(e^y + \cos z)$ по переменной x, затем по x и y, затем по x, y и z. Вот одно из возможных решений:

```
Rest[FoldList[D[#1,#2]&,x/(E^y + Cos[z]),{x,y,z}]]
```

$$\left\{\frac{1}{E^y + Cos[z]}, -\left(\frac{E^y}{E^y + Cos[z]}\right), -\left(\frac{2E^y Sin[z]}{(E^y + Cos[z])^2}\right)\right\}$$

Если к аргументу x нужно применить последовательно функции с заголовками f_1, f_2 и т.д., то в некоторых случаях это удобно делать с помощью встроенной функции **Composition**.

```
Composition[Cos,Sqrt,N][Pi]
-0.2000294
```

В примере к аргументу *Pi* сначала применяется функция **N**, затем **Sqrt**, а потом **Cos**. Посмотреть, как изменяется результат после применения очередной функции из композиции, можно, обратившись к **ComposeList**.

ComposeList[{N, Sqrt, Cos}, Pi]
{*Pi*, 3.14159, 1.77245, −0.200294}

Заметьте, что у функции **ComposeList** порядок функций изменен по сравнению с **Composition**.

При работе с алгеброй функций, когда функции естественно рассматривать как операторы, важную роль играет тождественный оператор **Identity**.

Identity[x]
x

Если f есть заголовок некоторой функции, то с помощью **InverseFunction** можно формально определить обратную функцию.

ff = InverseFunction[f]
$f^{(-1)}$

Формальность определения заключается в том, что, хотя композиции f и $f^{(-1)}$ являются тождественными функциями:

{Composition[ff, f][x], Composition[f, ff][y]}
$\{x, y\}$

с заголовком $f^{(-1)}$ не связываются автоматически никакие правила фактического вычисления обратной функции.

f[x_] := x^2
ff[y]
$f^{(-1)}[y]$

Таким образом, рассматриваемая алгебра функций скорее является алгеброй заголовков функций. Заголовки функций можно складывать, перемножать и умножать на вещественные числа, т.е. можно вводить объекты вида $f+g$, fg и $2f$ и т.п.

{(f+g)[x], (f g)[x], (2 f)[x]}
$\{(f+g)[x],\ (fg)[x],\ (2f)[x]\}$

Однако последний результат плохо согласуется с общепринятым определением суммы и произведения функций, которые определяются поточечно, т.е., например, суммой двух функций называется функция, значения которой при любых значениях аргументов равны сумме значений слагаемых. Функция **Through** обеспечивает такое согласование.

{Through[(f+g)[x]], Through[(f g)[x]], Through[(2 f)[x]]}
$\{f[x]+g[x],\ f[x]g[x],\ 2[x]f[x]\}$

В последнем выражении число 2 рассматривается как заголовок функции, а не как множитель. Выход из этого положения заключается в том, чтобы вместо чисел использовать чистые функции 2&, 3& и т.д., которые трактуются как функции, принимающие постоянные значения.

Through[(2&f)[x]]
$2f[x]$

5.4. Подмножества конечного множества

Программирование строго в функциональном стиле приводит к выражениям, получающимся нанизыванием применений заголовков функций к исходным аргументам. Такие выражения иногда называют „однострочниками“. В качестве примера однострочника приведем программу, которая по заданному конечному множеству символов дает список всех подмножеств

данного множества. Из всех подходов к решению задачи выберем алгебраический. Пусть задано множество

$$l = \{a, b, c, d\};$$

тогда все его подмножества можно отождествить со слагаемыми суммы, получающейся при раскрытии скобок произведения $(1+a)(1+b)(1+c)(1+d)$.

Expand[(1 + a)(1 + b)(1 + c)(1 + d)]
$1 + a + b + ab + c + ac + bc + abc + d + ad + bd + abd + cd + acd + bcd + abcd$

Естественно, единицу следует сопоставить с пустым подмножеством. Приведем последовательные этапы программирования этого метода и его окончательную форму.

l1 = 1 + l
$\{1 + a, 1 + b, 1 + c, 1 + d\}$

l2 = Times@@l1
$(1 + a)(1 + b)(1 + c)(1 + d)$

l3 = Expand[l2];
l4 = List@@l3
$\{1, a, b, ab, ac, bc, abc, d, ad, bd, abd, cd, acd, bcd, abc\}$

l5 = {#}&/@l4
$\{\{1\}, \{a\}, \{b\}, \{ab\}, \{ac\}, \{bc\}, \{abc\}, \{d\}, \{ad\}, \{bd\}, \{abd\}, \{cd\}, \{acd\}, \{bcd\}, \{abc\}\}$

l6 = l5 /. {Times → Sequence, {1} → {}}
$\{\{\}, \{a\}, \{b\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}, \{d\}, \{a, d\}, \{b, d\}, \{a, b, d\}, \{c, d\}, \{a, c, d\}, \{b, c, d\}, \{a, b, c\}\}$

Выражение **l6** дает ответ. Поставленный вместо **Times** заголовок **Sequence** превращает произведение в последовательность

аргументов, разделенных запятыми. Вот запись представленной программы в виде однострочника:

{#}& /@List @@ Composition[Expand, Times] @@ (1 + {a, b, c, d}) /. {Times → Sequence, {1} → {}}

Упражнения

1. Число называется совершенным, если оно равно сумме своих делителей, в число которых включается единица, но не само число. Например, число 6 совершенное. *Определите* функцию **perfectQ**, которая принимает значение True, если число совершенное, и False в противном случае. Воспользуйтесь встроенными функциями **Divisors** и **Apply**. С помощью функции **perfectQ** *проверьте*, что 28 совершенное число. *Найдите* следующее за 28 совершенное число.

2. С математической точки зрения игральная карта, скажем, дама пик, есть элемент прямого произведения двух множеств. Элементы первого множества есть масти: пики (spears), трефи (clubs), черви (hearts) и бубны (diamonds). Второе есть множество достоинств карт: туз (Ass), король (King), дама (Dame), валет (Jacket), 10 и далее до 7 (или до 2). С помощью функции **Outer** *получите* список CardDeck, элементы которого есть списки из двух элементов, в которых первым элементом являются символы s, c, h или d, а вторым — символы A, K, D, J, 10 и т.д. до 7. *Убедитесь*, что длина CardDeck равна 32.

3. Совершенным перемешиванием называется перемешивание колоды CardDeck, получаемое делением колоды на две равные части и последующим расположением карт частей одна за другой в чередующемся порядке. *Получите* такое перемешивание с помощью функций **Partition** и **Transpose**. *Напишите* функцию **perfectshuffle**, первым аргументом которой является колода (с произвольным расположением карт), а вторым аргументом — натуральное число n и которая осуществляла бы совершенное перемешивание n раз подряд. *Убедитесь*, что после пяти совершенных перемешиваний списка CardDeck восстанавливается исходное расположение карт. *Напишите* функцию **perfectshuffleright**, отличающуюся от предыдущей тем, что прежде чем применять функцию **Transpose**, следует переставить местами первую и вторую части колоды. *Убедитесь*, что в данном случае исходное расположение карт восстанавливается только после десяти перемешиваний.

4. *Напишите* функцию **deleterand**, которая удаляла бы из данного множества n элементов случайным образом. Воспользуйтесь для этого

встроенными функциями **Delete**, **Random**, **Nest**. Воспользуйтесь **deleterand** для того, чтобы написать функцию **deal**, осуществляющую раздачу четырем игрокам карт из CardDeck.

5. Встроенная функция **IntegerDigits**[n,b], где n — число, b — база системы исчисления, порождает список цифр в представлении числа n в системе с базой b. Так, при $n = 12$, $b = 2$ имеем **IntegerDigits**[**12**, **2**] = {**1**, **1**, **0**, **0**}. С помощью функции **Fold** *напишите* функцию **tobase** от двух аргументов, которая записывала бы число n, заданное в десятичной системе, в системе исчисления с базой b при условии, что $b \leq n$.

6. *Получите* список повторяющихся в списке l элементов.

7. *Получите* список позиций, которые занимают простые числа в списке $l =$ **Range**[10].

8. По списку $l = \{a, b, a, b, c, d, d, a\}$ *получите* новый список, элементами которого были бы пары вида { элемент l, число его вхождений в l }.

9. Дан список

   $l = \{\{1,1,1,1\},\{1,2,3,4\},\{1,0,2,0\},\{1,3,0,7\}\}$

   векторов, образующих базис в четырехмерном линейном пространстве. *Напишите* однострочник для получения матрицы Грама gs этих векторов, матричные элементы которой суть попарные скалярные произведения векторов базиса.

10. Используя список l и матрицу Грама gs предыдущей задачи, *перейдите* от базиса l к новому базису, четвертый вектор $v[4]$ которого ортогонален трем первым $v[1]$, $v[2]$ и $v[3]$ и нормирован на единицу. Для этого *замените* $v[4]$ на линейную комбинацию $v = v[4] + x\,v[1] + y\,v[2] + z\,v[3]$ с подходящими x, y, z.

Глава 6
ПРОГРАММИРОВАНИЕ, ОСНОВАННОЕ НА ПРАВИЛАХ ПРЕОБРАЗОВАНИЙ

Понятие преобразования является, по-видимому, одним из самых общих в математике. Таблицу умножения можно рассматривать как совокупность правил преобразований, в соответствии с которыми не будет ошибкой заменить выражение 2×2 всюду, где оно встречается, на 4, выражение 2×3 — на 6 и т.д. Алгебраические вычисления проводятся на основе тождеств, которые также можно рассматривать как правила преобразований. Например, тождество $(x+y)^2 = x^2 + 2xy + y^2$ позволяет всюду в алгебраических формулах заменить его левую часть на правую.

В математическом анализе вычисления часто сводятся к манипулированию выражениями, содержащими интегралы, суммы, производные и т.п., согласно справочным руководствам, содержащим значения этих интегралов, сумм и т.д., которые подставляются на место исходных выражений с тем, чтобы упростить или получить численные значения рассматриваемых выражений. Так, вместо интеграла $\int_{-\infty}^{\infty} e^{-x^2}\,dx$ можно подставить число $\sqrt{\pi}$.

Другой аспект правил преобразований заключается в том, что их можно рассматривать как аксиомы, которым удовлетворяют определенные математические объекты. Операция сложения ассоциативна, и это свойство можно записать в виде правила $(a+b)+c = a+(b+c)$, операция дифференцирования линейна: $(af(x)+bg(x))' = af'(x)+bg'(x)$, при a и b постоянных, и удовлетворяет правилу Лейбница: $(f(x)g(x))' =$

$f'(x)g(x) + f(x)g'(x)$. „Математика“ имеет развитую систему средств, позволяющих создавать весьма изощренные правила преобразований. На них основана самостоятельная парадигма программирования, с которой можно познакомиться в этой главе.

6.1. Глобальные и локальные правила преобразований

Выражение $x = a$, трактуемое в традиционных языках программирования как присвоение значения a символу x, можно рассматривать как простейшее правило преобразования, согласно которому всюду, т.е. во всех выражениях „Математики“, в которые входит символ x, его следует заменить на a. В этом смысле рассматриваемое правило глобально. Его локальный аналог есть подстановка $x \to a$, которая осуществляется в отдельных формулах после применения к ним функции **ReplaceAll**, имеющей вид /. во входном формате.

Знак = является инфиксной формой функции двух аргументов **Set**. Выражение **Set[expr$_1$,expr$_2$]** вычисляется следующим образом. Аргумент *expr*$_1$ вовсе не вычисляется, выражение *expr*$_2$ вычисляется, и это вычисленное значение присваивается выражению *expr*$_1$. Подобный порядок вычисления имеет свои основания. Например, он позволяет избежать следующего противоречия. Пусть сначала символу x присвоено значение 1, т.е. вычислено выражение **x = 1**, а затем возникла необходимость присвоить символу x новое значение 2, вычислив **Set[x, 2]**. Если бы первый аргумент функции **Set** вычислялся, то возникла бы ситуация, когда числу 1 присваивалось бы значение 2. Подчеркнем, что вычисленное выражение *expr*$_2$ в принципе может быть присвоено в качестве значения любому выражению *expr*$_1$, а не только символу.

a[x^2] = E^x
E^x

После этого всюду вместо $a[x^2]$ будет подставляться E^x. Попробуем, однако, определить следующее правило преобразования:

x^2 = b
Set:: write: Tag Power in x^2 *is Protected*
b

Мы получили сообщение, что заголовок **Power** в выражении x^2 обладает атрибутом Protected. Это означает, что с выражениями x^2, x^3, $Sqrt[x]$ и любыми выражениями Power[$expr_1, k$] с заголовком Power нельзя без специальной процедуры, описываемой ниже, связать какое-либо правило преобразования с помощью функции **Set**. Пример подсказывает также, что „Математика" ассоциирует преобразования с заголовком выражения $expr_1$ в **Set[expr₁, expr₂]**. Это обстоятельство позволяет определять новые функции, вычисляя выражения вида

f[x_] = x^3
x^3

в которых символ x сопровождается подчеркиванием _, являющимся входной формой выражения **Blank[ ]**. После этого $f[expr]$ для любого $expr$ будет заменяться на $expr^3$, и это правило преобразования будет прочно связано с заголовком f, составляя одно из так называемых *нижних значений* этого заголовка.

DownValues[f]
$\{Literal[f[x_]] :> x^3\}$

С любым заголовком можно ассоциировать сколько угодно нижних значений, или, что то же самое, правил преобразований. Например, из определенной выше функции $f(x) = x^3$ можно сконструировать разрывную функцию, переопределив ее значение в точке 0:

f[0] = 1
1

DownValues[f]
$\{Literal[f[0]] :> 1, Literal[f[x_]] :> x^3\}$

Отметим, что „Математика“ будет применять правила преобразований, связанные с символом **f**, в том порядке, в котором они даны в DownValues. *Всегда более частные правила предшествуют более общим.* Таким образом, при вычислении выражения **f[0]** будет получен результат 1, а не 0.

Помимо функции **Set** для определения правил преобразований используется функция **SetDelayed**, или отложенное присвоение, входной формой которой является $\mathbf{expr_1 := expr_2}$. Вычисление выражения **SetDelayed[**$\mathbf{expr_1, expr_2}$**]** заключается в том, что ни $expr_1$, ни $expr_2$ не вычисляются, но соответствующее правило заносится в список нижних значений заголовка выражения $expr_1$. Если в дальнейшем возникнет необходимость вычислить выражение $expr_1$, то оно будет заменяться на вычисленное именно в тот момент значение выражения $expr_2$. Разницу между немедленным и отложенным присвоением можно понять, обратившись к следующим примерам:

x = Random[ ]
0.0489537

y := Random[ ]
{Sin[x], Sin[x], Sin[y], Sin[y]}
{0.0489341, 0.0489341, 0.758444, 0.311081}

В примере **Sin[x]** вычислялся для фиксированного значения x, равного 0.0489537, а **Sin[y]** — для y, являющихся значениями функции **Random** в момент вычисления функции **Sin**.

Map[ArcSin[#]&, %]
{0.0489537, 0.0489537, 0.860922, 0.31633}

Если этот пример показался читателю экзотическим, то следующий описывает довольно обычную ситуацию в символьных алгебраических вычислениях. Определим две функции $h(x)$ и $hh(x)$ следующим образом:

```
h[x_] = Factor[1 + x^2]
1 + x^2
hh[x_] := Factor[1 + x^2]
{h[Iy], hh[Iy]}
{1 - y^2, (1 - y)(1 + y)}
```

Функция $h(x)$ тождественно равна $1+x^2$, поэтому $h(iy) = 1 + (iy)^2 = 1 - y^2$, в то время как при вычислении значения $hh(iy)$ выполняется разложение на множители многочлена $1 - y^2$.

```
{DownValues[h], DownValues[hh]}
{{Literal[h[x_]] :> 1 + x^2}, {Literal[hh[x_]] :> Factor[1 + x^2]}}
```

Если заголовок функции, с которой желательно ассоциировать новые правила преобразований, имеет атрибут Protected, то этот атрибут можно убрать с помощью встроенной функции **Unprotect**, после чего не будет препятствий для определения новых ассоциированных правил.

```
Unprotect[Power]
{Power}
x^2 = b
b
Protect[Power]
{Power}
f[x^2]
b^3
```

Тот факт, что многие встроенные функции снабжены атрибутом Protected имеет свои веские причины. Дело в том, что вычисление выражений включает просмотр „Математикой" большого числа правил преобразований, ассоциированных с их заголовками. Поскольку такие встроенные функции, как **Power**, **Plus**, **Times** и т.д., весьма часто встречаются в вычислениях, то каждое новое правило, ассоциированное с ними, может существенно замедлить работу „Математики". По этой же причине желательно снова запротектировать такие функции, если пользователь все же сочтет необходимым однажды распротектировать их.

Имеется другой механизм определения правил преобразований для выражений с протектированными заголовками, а именно механизм *верхних значений* (**UpValues**). Функции **UpSet** и **UpSetDelayed**, входные формы которых есть ^ = и ^ :=, позволяют ассоциировать правила преобразований с элементами, находящимися на первом уровне выражений, имеющих любые, в том числе и протектированные, заголовки.

y^p^ = c
c

?y
Global`y
y/ : y^p = c

?p
Global`p
p/ : y^p = c

Итак, новое правило преобразования, согласно которому выражение y^p заменяется на c, ассоциировано с символами y и p, составляя их верхние значения. С учетом правил преобразования, связанных с верхними значениями, имеем следующий результат вычисления:

f[y^p]
c^3

Попробуем воспользоваться функцией **UpSet**, чтобы определить правило $x^3 = d$:

x^3^ = d
Set:: write: Tag Integer in x^3 is Protected
d

Оказывается, правило можно ассоциировать с символами и нельзя ассоциировать с числами и строками. Однако можно указать „Математике“, чтобы она ассоциировала новое правило только с символом x:

x/ : x^3 = d
d

f[x^3]
d^3

Верхние значения позволяют строить простейшие базы данных, представление о которых можно получить из следующего примера.

Ivanov / : age[Ivanov] = 30; Ivanov / : sex[Ivanov] = male;
Ivanov / : occupation[Ivanov] = engineer;

Ivanchuk / : age[Ivanchuk] = 33;
Ivanchuk / : sex[Ivanchuk] = female;
Ivanchuk / : occupation[Ivanchuk] = accountant;

Выше введены сведения о двух работниках: Иванове и Иванчуке. Теперь можно получить сведения о возрасте, поле и служебном положении каждого из них, сделав хотя бы такой запрос:

?Ivanov
Global‘Ivanov
$age[Ivanov]\hat{} = 30$
$sex[Ivanov]\hat{} = musc$
$occupation[Ivanov]\hat{} = engineer$

Как уже отмечалось, локальным аналогом глобального правила подстановки, определяемого с помощью функции **Set**, является **Rule[expr₁,expr₂]**, или в инфиксной форме $\mathbf{expr_1 \to expr_2}$. Для функции **SetDelayed** аналогичное ей локальное правило задается с помощью функции **RuleDelayed[expr₁,expr₂]**, или во входном формате $\mathbf{expr_1 :> expr_2}$. Оба вида локальных правил применяются с помощью функции **ReplaceAll**.

rl1 = x^2 → −y^2
$x^2 \to -y^2$
{h[x]/.rl1, hh[x]/.rl1}
$\{1 - y^2, 1 - y^2\}$

В обоих случаях правило, или подстановка, **rl1** применяется к уже вычисленным выражениям **h[x]** и **hh[x]**, поэтому результаты вычислений совпадают.

Оба типа локальных правил могут быть использованы с любыми шаблонами. Пусть в процессе вычислений встретилось выражение

ex1 = x^((n + 1)/n − 1/n)
$x^{(\frac{n+1}{n} - \frac{1}{n})}$,

которое после приведения выражения в показателе к общему знаменателю равно x. „Математика“ сама по себе этого преобразования не выполняет, поэтому попытаемся заставить ее выполнить преобразование с помощью локальных правил:

{rl2 = x^k_ → x^Together[k], rl3 = x^k_ :> x^Together[k]}
$\{x^{k_} \to x^k, x^{k_} :> x^{Together[k]}\}$

{ex1 /. rl2, ex1 /. rl3}

$\{x^{(n+1)/n-1/n}, x\}$

В этом случае, как и следовало ожидать, эффективным оказалось правило **rl3**.

6.2. Шаблоны

Именованные шаблоны вида *x_* использовались нами при определении новых функций. Начнем систематическое знакомство с этим понятием. В принципе шаблоны используются для выделения классов выражений „Математики“. Самый общий шаблон имеет вид _, соответствующий класс состоит из всех выражений „Математики“. Конкретные выражения $h[e_1, \ldots, e_n]$ являются самыми узкими шаблонами, выделяя классы, состоящие из одного выражения. Остальные шаблоны по степени общности занимают промежуточное положение. Рассмотрим, например, выражение x^3, или в полной форме *Power*[*x*,3]. Ясно, что оно является единственным элементом класса, выделяемого этим же выражением, и входит в класс выражений, определяемых шаблоном _. Кроме того, оно входит в класс выражений вида *x*^_, который можно описать как „*x* в любой степени“. Классы выражений _^3, „нечто в третьей степени“, и _^_, „нечто в любой степени“ также содержат x^3. Будем говорить, что некоторое выражение *отвечает шаблону*, если оно входит в определяемый этим шаблоном класс.

Шаблон _ можно снабдить именем, которое должно быть символом. Именованный шаблон _ имеет вид *symbol_*. Имя шаблона не влияет на класс выражений, им выделяемых. Кроме того, можно указать заголовок шаблона в форме _*head*. Заголовок существенно сужает класс выражений. Так, шаблон _*Integer* выделяет класс атомарных выражений, являющихся целыми числами. Рассматриваемое нами выражение x^3 отвечает шаблону *y*_^3 и не отвечает шаблону *x*^_*Real*.

Имеется встроенная функция **MatchQ**, проверяющая соответствие выражений шаблонам. Выражение **MatchQ[expr, pattern]** принимает значение True, если выражение *expr* отвечает шаблону *pattern*, и принимает значение False — в противном случае.

{MatchQ[x^3, _Symbol^_Integer],
MatchQ[x^3, x^(_Integer + _Integer)]}
{True, False}

Последний пример достаточно поучителен. С точки зрения обычной математики любое целое число всегда можно представить в виде суммы двух других целых чисел, и сумма целых чисел есть снова целое число. Поэтому можно было бы ожидать, что x^3 отвечает шаблону $x\hat{}(_Integer + _Integer)$. На самом деле это не так, потому что „Математика" сравнивает внутренние полные формы выражения и шаблона, а не устанавливает их математическую эквивалентность.

Кроме шаблонов, основу которых составляет подчеркивание _ (**Blank[ ]**), имеются шаблоны вида __ — двойное подчеркивание (**BlankSequence[ ]**) и тройное подчеркивание ___ (**BlankNullSequence[ ]**). Двойное подчеркивание выделяет класс выражений, состоящих из одного или из нескольких выражений „Математики", разделенных запятыми. Например, выражение **h[e1, e2, e3]** отвечает шаблону *h[x__]*. Тройное подчеркивание выделяет класс, состоящий из нуля или из нескольких выражений, разделенных запятыми.

MatchQ[{a, b, c}, {a, ___, b, ___}]
True

Аналогично шаблону _ эти последние шаблоны можно снабжать именами. Шаблоны __*head* и ___*head* предполагают, что все выражения последовательности имеют одинаковый заголовок.

MatchQ[{a, b, 2}, {__Symbol}]
False

Возвращаясь к шаблонам, основанным на одном знаке _, отметим, что можно ограничивать класс выражений, определяемых шаблоном, не только фиксируя заголовок шаблона, но и требуя выполнения определенного условия, выраженного предикатом. Например, в класс выражений, выделяемых шаблоном **_?EvenQ**, попадают только четные целые числа. Наряду со встроенными предикатами могут быть использованы предикаты, определяемые пользователем с помощью анонимных функций.

MatchQ[Sin[x] + Cos[y] + z, _?(MemberQ[#, Sin[x]]&)]
True

Рассмотрим теперь шаблоны, которые содержат альтернативу. На самом деле такие шаблоны представляют собой совокупность нескольких шаблонов, соединенных знаком |, играющим роль логической связки ИЛИ.

MatchQ[a, a|b]
True
MatchQ[y^2 + 1, a_.x_^2 + b_.x_ + c_.]
False
MatchQ[y^2 + 1, a_.x_^2 + b_.x_ + c_.|a_.x_^2 + c_.]
True

В двух вышеприведенных примерах нам встретились шаблоны вида **a_.x_^2**, в которых вслед за символом _ стояла точка. Это пример шаблонов со значениями *по умолчанию*. В рассматриваемом шаблоне сомножитель **a_** может отсутствовать, а на его месте по умолчанию будет поставлен множитель 1. Вот почему выражение *y^2 + 1* отвечало шаблону *a_.x_^2 + c_.*. Шаблон *c_* также сопровождался точкой, и его значение по

умолчанию равно 0. Итак, в шаблоне x_+y_. значение по умолчанию шаблона, сопровождаемого точкой, равно 0, в шаблонах x_y_. и x_^y_. значение по умолчанию равно 1. Пользователь может по своему желанию присвоить шаблонам значения по умолчанию. Для этого он может воспользоваться конструкцией: x_ : v, означающей, что если соответствующий шаблон или часть шаблона отсутствуют, то они заменяются на выражение *v*.

```
f[x_,y_:5] := x+y
{f[a,b], f[a]}
{a+b, 5+a}
```

Встречаются случаи, когда необходимо дать имя не только шаблону вида _, но и более общему шаблону, содержащему несколько подчеркиваний _. Имя такого шаблона помещается перед всем шаблоном и отделяется от него двоеточием, т.е. используется конструкция *x* : *pattern*.

```
f[x^2]+f[y^(1/2)]/.f[a:_^_Integer] :> g[a]
f[Sqrt[y]]+g[x^2]
```

В заключение отметим, что шаблоны доставляют еще один инструмент работы со списками, а именно имеется возможность выделять элементы списков не только по их позициям или по свойствам, которыми они обладают (функции **Part** или **Select**), но и по соответствию их шаблонам. Функция **Cases** выбирает, а функция **DeleteCases** удаляет из списка элементы, отвечающие шаблону *pattern*.

```
{Cases[{x y,x+y,x^y},a_b_],
DeleteCases[{x y,x+y,x^y},a_b_]}
{{x y},{x+y,x^y}}
```

6.3. Шаблоны в глобальных правилах преобразований

Использование шаблонов в правилах преобразований как глобальных, так и локальных, вооружает программиста гибким средством, позволяющим создавать короткие по длине, но весьма содержательные программы. Этот параграф будет состоять в основном из примеров применения шаблонов.

Допустим, что нужно определить функцию $h(x)$ вещественной переменной x, равную 0 при $x < 0$ и равную x при $x > 0$. В определении этой функции участвуют два предиката, поэтому в определении вида **h[x_] := expr** приходится использовать два выражения *expr* в зависимости от выполнения того или иного условия. Это ограничение действия определения достигается с помощью функции /;, являющейся инфиксной входной формой функции **Condition**:

```
h[x_] := 0/;x < 0
h[x_] := x/;x >= 0
```

Для того чтобы убедиться в успешном решении задачи, построим график функции $h(x)$ (рис. 6.1):

```
Plot[h[x], {x, -1, 1}];
```

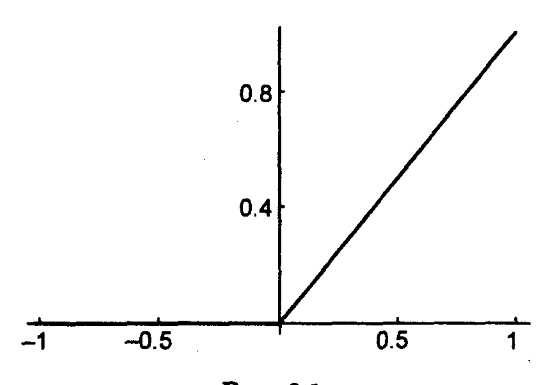


Рис. 6.1

К сожалению, определенную в такой форме функцию $h(x)$ нельзя ни продифференцировать, ни проинтегрировать в конечных пределах:

D[h[x],x]
h'[x]

Integrate[h[x],{x,1,2}]
Integrate[h[x],{x,1,2}]

поэтому дадим отдельные определения для этих операций с функцией $h(x)$. Функция **D** является протектированной, следовательно, для определения производной доступен механизм верхних значений. Поскольку производная от $h(x)$ равна функции $\theta(x)$, принимающей значение 0 при $x < 0$ и 1 при $x > 0$, то определим сначала эту функцию:

theta[x_] := 0/;x < 0; theta[x_] := 1/;x >= 0

а затем и производную от $h(x)$:

h / : D[h[x_],x_] := theta[x]

Произведем вычисления производной:

{D[h[x],x], D[h[x],x]/.x → −2, D[h[x],x]/.x → 10}
{*theta*[*x*], 0, 1}

Аналогичное определение может быть дано и для интеграла от функции $h(x)$.

В качестве второго примера определим функцию **log**, вычисляющую логарифмы вещественных чисел по основанию 2 в предположении, что встроенная функция **Log[2,x]** нас по каким-то причинам не устраивает. Действительно, встроенная функция **Log** по умолчанию определена для всех, в том числе и

комплексных значений аргумента, поэтому она не удовлетворяет некоторым тождествам, справедливым только в вещественном случае. Например, $\mathbf{Log[E^z]} \neq \mathbf{z}$ и т.д. Для того чтобы выполнить поставленную задачу, можно дать следующую последовательность правил преобразований:

```
log[2] = 1;  log[1] = 0;  log[y_^x_] := x log[y];
log[x_ y_] := log[x] + log[y]
```

фиксирующих основные свойства логарифмической функции. Вычислим выражение

```
log[2^a b/Sqrt[c]]
```

$$a + log[b] - \frac{log[c]}{2}$$

Результат соответствует нашим ожиданиям. Чтобы иметь возможность находить численные значения функции **log** для вещественных значений аргумента, добавим правило

```
log[x_Real] := FixedPoint[((# - (2^# - x)/
(2^#Log[2.]))&, 1.]
```

основанное на методе итераций Ньютона.

```
{log[3.], Log[2, 3.]}
{1.58496, 1.58496}
```

Вполне удовлетворительная точность для чисел порядка 1. Определим правило вычисления производной для функции **log** способом, отличным от только что использованного. Для этого воспользуемся тем обстоятельством, что функция **D** в случае, когда не имеется правил для вычисления производной от данной функции $f(x)$, имеет результатом выражение, полная форма которого есть **Derivative[1][f][x]**. Функция **Derivative[1]** не протектирована, ее аргументом являются заголовки функций, поэтому определим производную от **log** следующим образом:

Derivative[1][log] := (1/(#Log[2])&
D[log[xf[x]],x]

$$\frac{1}{x Log[2]} + \frac{f'[x]}{f[x] Log[2]}$$

Неопределенный интеграл от функции **log** может быть определен с помощью верхних значений:

log / : Integrate[log[x_],x_] := x log[x] – x/Log[2]

Хорошо известно, что функция $fact[n] = n!$ от целого неотрицательного аргумента n удовлетворяет следующему тождеству: $fact[n] = n fact[n-1]$, которое дает возможность последовательно определять значения этой функции, отправляясь от значения $fact[1] = 1$. Рассматриваемое тождество является простейшим рекурсивным определением функции. Более сложным является рекурсивное определение функции $fib[n]$, чьими значениями являются числа Фибоначчи: $fib[n] = fib[n-1] + fib[n-2]$, дополненное значениями $fib[1] = 1$, $fib[2] = 1$. Рассматриваемые тождества могут быть непосредственно положены в основу определения соответствующих функций:

fact[n _Integer?Positive] := n fact[n – 1]
fact[1] = 1
{fact[5], fact[–3], fact[3.1]}
$\{120, fact[-3], fact[3.1]\}$

Функция $fact[x]$ вычисляется в соответствии с данными правилами для целых положительных x, а для других значений аргумента не определена.

fib[n _Integer?Positive] := fib[n – 1] + fib[n – 2]
fib[1] = fib[2] = 1;
fib[6]
8

Рекомендуем читателю посмотреть, прибегнув к функции **Trace**, как вычислялся последний результат.

Оказывается, „Математика“ последовательно применяла основное правило, понижая значение аргумента до тех пор, пока не пришла к отдельно определенным начальным значениям $fib[1]$ и $fib[2]$, а затем проводила вычисления для возрастающих значений аргумента. При таком порядке вычислений нахождение $fib[5]$ или $fib[7]$ выполняется совершенно аналогично, так как значения функции $fib[n]$ не запоминаются. Можно исправить это положение вещей, прибегнув к так называемому „динамическому программированию“. Это достигается следующим изменением данных ранее правил.

```
factdyn[n _Integer?Positive] :=
factdyn[n] = nfactdyn[n - 1]
factdyn[1] = 1;
```

Теперь все вычисляемые значения функции **factdyn** будут запоминаться.

```
factdyn[5]
120
```

Сравним время вычисления 100! обеими функциями.

```
{fact[100]; // Timing, factdyn[100]; // Timing}
{{0.22Second, Null}, {0.61Second, Null}}
```

Время, затраченное второй функцией, существенно больше. А теперь для числа 110! сравним вновь время его вычисления обеими функциями.

```
{fact[110]; // Timing, factdyn[110]; // Timing}
{{0.27Second, Null}, {0.11Second, Null}}
```

Результат достаточно убедителен. Экономия времени в случае применения функции **factdyn** достигнута за счет того, что значение *factdyn*[100] сохранено в программе.

С понятием опций, т.е. необязательно явно указываемых аргументов функций, мы познакомились в гл. 3 при изучении встроенных графических функций. Там опциональные аргументы указывали директивы, регулирующие представление графического объекта. В „Математике“ имеются два способа определять функции с опциями. Первый основан на использовании именованных шаблонов **x_ : v** со значениями по умолчанию равными *v*; второй — на использовании функции **Options** и локальных правил преобразований.

Рассмотрим эти способы на примере. Допустим, что при работе со списками нам часто требуется разбивать элементы списков на пары с отступом 1, и мы пользуемся для этого функцией **Partition**, указывая в качестве ее второго аргумента число 2, а в качестве третьего — число 1. Чтобы сэкономить время, определим функцию **pairs**, позволяющую не указывать явно числа 2 и 1. Сделаем это упомянутыми двумя способами:

```
pairs[x_List, n_ : 2, m_ : 1] := Partition[x, n, m]
pairs[{a, b, c, d}]
{{a,b},{b,c},{c,d}}
```

Функцией **pairs**, учитывая ее определение, можно пользоваться так же, как и функцией **Partition**, указывая явно один или два ее необязательных аргумента. Следует помнить, что если будет указан один аргумент, то он будет воспринят как второй аргумент функции **pairs**. Пусть, например, требуется разбить список на тройки с отступом 1. Воспользуемся функцией **pairs** следующим образом:

```
pairs[{a, b, c, d}, 3]
{{a,b,c},{b,c,d}}
```

Если же требуется разбить список на пары с отступом два, то следует вычислить выражение:

pairs[{a, b, c, d}, 2, 2]
{{a,b}, {c,d}}

Второй способ реализуется следующим образом:

Clear[pairs]
Options[pairs] = {opt1 → 2, opt2 → 1}
{opt1 → 2, opt2 → 1}

pairs[x _List, opts ___] := Partition[x, opt1 /.
{opts} /. Options[pairs], opt2 /. {opts} /. Options[pairs]]
pairs[{a, b, c, d}]
{{a,b}, {b,c}, {c,d}}

При втором способе определения функции **pairs**, чтобы решить задачу о разбиении рассматриваемого списка на тройки, следует вычислить выражение:

pairs[{a, b, c, d, }, opt1 → 3]
{{a,b,c}, {b,c,d}}

6.4. Шаблоны в локальных правилах преобразований

Приемы программирования, рассматриваемые в этом параграфе, являются, на наш взгляд, одними из самых интересных, неожиданных и эффективных из используемых в „Математике“. Можно надеяться, что с течением времени программы в стиле локальных преобразований займут ведущее место в приложениях. Локальные правила содержат подстановки вида **expr1 → expr2** и **expr1 :> expr2**, где **expr1** может содержать шаблоны. Подстановки осуществляются с помощью функций

/., т.е. **ReplaceAll**, и //., т.е. **ReplaceRepeated**. На применение шаблонов, как и в глобальных правилах преобразований, можно накладывать ограничения с помощью указания заголовков шаблонов или с помощью предикатов.

Присутствие шаблонов, содержащих подчеркивание _, приводит к некоторым особенностям вычисления функций **ReplaceAll** и **RepalceRepeated**, которые необходимо обсудить. Они относятся к тем случаям, когда рассматриваемые функции применяют правила подстановок, содержащие списки подстановок.

```
{i,j} /. {i → 1, j → 2}
{1,2}
```

Результат этого вычисления достаточно очевиден. Следующее вычисление менее очевидно.

```
{i,j} /. {i → 1, j → 2, {i,j} → 0}
0
```

Полученный ответ заставляет предположить, что шаблон $\{i, j\}$ в целом имеет большее соответствие с преобразуемым выражением, чем его отдельные части. Однако это правило работает отнюдь не во всех случаях.

```
{i,i} /. {{x_,x_} → 1, {__} → 0}
1
```

Этот результат как будто подтверждает правило, потому что шаблон {x_, x_} определяет более узкий класс выражений, чем шаблон {__}. Тем не менее если мы изменим порядок шаблонов в подстановке, то получим другой ответ.

```
{i,i} /. {{__} → 0, {x_,x_} → 1}
0
```

Следовательно, если требуется запрограммировать функцию, называемую δ-символом Кронекера: $\delta(i,j) = 1$ при $i = j$ и $\delta(i,j) = 0$ в противном случае, то следует написать определение:

```
delta[i_,j_] := {i,j} /. {{x_,x_} → 1, {__} → 0}
{delta[2,2], delta[1,2]}
{1, 0}
```

Определенные особенности имеет также и вычисление функции **ReplaceRepeated** с шаблонами. Вообще говоря, ее вычисление сводится к повторному вычислению функции **ReplaceAll** до тех пор, пока ни одно из подвыражений преобразуемого выражения не удается преобразовать в соответствии с хотя бы одним шаблоном списка. При этом порядок применения шаблонов следующий. Первый шаблон списка используется, пока в преобразуемом выражении находится хотя бы одно подвыражение, ему соответствующее. Затем применяется второй шаблон списка и т.д. Ни в коем случае не следует думать, что каждый раз применяется более чем один шаблон списка. Рассмотрим иллюстрирующий пример:

```
{1,2,a}//.{{_,x___,a} → {x,a},{___,2,___} → 2,{a} → a}
a
```

Мы видим, что первый шаблон списка применяется два раза, сводя выражение к одноэлементному списку $\{a\}$, а затем применяется третий шаблон. Второй шаблон не работает после первого применения **ReplaceAll**, хотя в этот момент исходное выражение преобразовано к виду $\{2, a\}$. После применения третьего шаблона полученное выражение a уже не соответствует ни одному из шаблонов, поэтому вычисления прекращаются.

Контроль за вычислением **ReplaceRepeated** может осуществляться с помощью условий, выраженных предикатами. В качестве примеров рассмотрим имитацию с помощью локальных правил преобразований некоторых функций, используемых

для генерации и преобразования списков. Начнем с функции **Range[n]**, порождающей список натуральных чисел от 1 до n. Идея состоит в том, чтобы, применяя подстановку $\{x__\} :> \{x, Length[\{x\}]+1\}$, с помощью функции **ReplaceRepeated** и стартуя со списка $\{1\}$, преобразовать последний в список $\{1,2,\ldots,n\}$.

```
{1}//.{x__} :> {x,Length[{x}]+1}/;Length[{x}] < 3
{1,2,3}
```

Конструкция прекрасно работает и позволяет определить функцию

```
range[n_] := {1}//.{x__} :> {x,Length[{x}]+1}/;
Length[{x}] < n
range[10]
{1,2,3,4,5,6,7,8,9,10}
```

Аналогичная идея лежит в основе имитации функции **Drop**.

```
drop[l_List,n_Integer] := l//.{x_,y___} :> {y}/;
Length[{y}] > Length[l] - n - 1
{drop[{a,b,c,d,e},3],drop[{a,b,c,d,e},5]}
{{d,e},{}}
```

Следующая задача предлагалась на соревновании по элегантному программированию на Первой конференции пользователей „Математики" в 1991 г. Дан список из повторяющихся чисел, например

```
l = {1,1,1,2,2,3,3,3,3,1,1};
```

Требуется составить список пар, первым элементом которых были бы элементы списка l, а вторыми — числа их смежных повторений. Таким образом, список l должен быть преобразован в список

{{1,3},{2,2},{3,4},{1,2}}

Вот одно из лучших решений задачи. Сначала список *l* преобразуется в список

l1 = {#,1}&/@l
{{1,1},{1,1},{1,1},{2,1},{2,1},{3,1},{3,1},
{3,1},{3,1},{1,1},{1,1}

а затем применяется последовательность локальных преобразований:

l1//.{x___,{a_,k_},{a_,m_},y___} :> {x,{a,k+m},y}
{{1,3},{2,2},{3,4},{1,2}}

Еще одним примером, позволяющим понять работу **ReplaceRepeated**, будет сортировка по убыванию численного списка. Пусть требуется расположить элементы списка

l = {2,5,3,1,4};

так, чтобы последующий элемент был строго меньше предшествующего. Вот решение этой задачи с помощью локальных правил:

l//.{x___,a_,b_,y___} :> {x,b,a,y}/;a < b
{5,4,3,2,1}

Упражнения

1. Пусть **l = {{0,a,b,c},{0,1,2},{0,x},{0}}** есть список, представляющий собой верхнюю треугольную часть кососимметрической матрицы **m** четвертого порядка. *Реализуйте* следующий способ восстановления матрицы **m** по списку **l**. Сначала с помощью локальных правил преобразований получите из **l** список **l1**, дополнив спереди нулями элементы **l** так, чтобы **l1** стала матрицей четвертого порядка. Затем получите **m** как **l1 − Transpose[l1]**.

2. Вронскианом семейства функций $\{f_1, f_2, \ldots, f_n\}$ от одного аргумента называется детерминант матрицы $\{\{f_1, \ldots, f_n\}, \{f_1', \ldots, f_n'\}, \ldots, \{f_1^{(n)}, \ldots, f_n^{(n)}\}\}$. *Определите* функцию **wronsk** от двух аргументов, первый из которых есть список функций, второй — аргумент функций, вычисляющую вронскиан.

3. *Определите* функцию **ord**, которая, будучи примененной к частной производной функции с произвольным заголовком, т.е. к выражению вида

   Derivative[n1,n2,...][f][x,y,...],

   имела бы значением порядок производной. Дать по крайней мере два решения: один в функциональном стиле, другой в стиле локальных правил преобразований.

4. Используя локальные правила преобразований, *дайте определение* функции **newton**, которая осуществляет процесс итераций для нахождения корней функций с заданной точностью. Функция должна иметь три аргумента: заголовок функции, начальное приближение и точность.

5. *Определите* функцию **crossproduct**, которая по заданным векторам трехмерного пространства вычисляла бы их векторное произведение.

6. *Напишите* определение для функции **dirderiv**, которая по заданному вектору трехмерного пространства и функции, определенной на этом пространстве, вычисляет производную функции по направлению вектора. *Считайте* систему координат фиксированной, поэтому для задания функции достаточно ее заголовка.

7. С помощью локальных правил *определите* функцию **decode**, преобразующую списки вида $\{\{a,2\},\{b,5\},\ldots\}$ в списки $\{a,a,b,b,b,b,b\ldots\}$.

8. *Определите* функцию **subsets**, порождающую все подмножества заданного непустого множества. *Реализуйте* следующую конструкцию: при фиксированном произвольном элементе непустого множества все его подмножества распадаются на два класса. В первый класс попадают все подмножества, содержащие фиксированный элемент, во второй класс — все остальные.

9. *Определите* независимо от функции **subsets** функцию **ksubsets**, порождающую все k-элементные подмножества заданного непустого множества. Определение должно быть основано на той же конструкции, что и в случае функции **subsets**.

Глава 7
ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Несмотря на то что наиболее естественными для „Математики“ стилями программирования являются функциональный стиль и стиль определений, или правил преобразований, „Математика“ содержит средства и конструкции, позволяющие программировать в традиционном процедурном стиле языков Фортран, Паскаль, Бэйсик и им подобных. Кроме того, некоторые функции процедурного программирования, такие, как **If**, **Which**, часто используются и вне процедурного стиля. В принципе все программы процедурного стиля имеют свои аналоги в упомянутых ранее стилях, причем гораздо более эффективно отлаживаемые, выполняемые и понимаемые. Средства процедурного программирования, таким образом, могут рассматриваться как уступка тем пользователям, которые привыкли к процедурным языкам, с целью облегчить им освоение языка программирования „Математика“.

7.1. Составные выражения. Оператор Do

Одним из основных операторов процедурного программирования является оператор присвоения. В „Математике“ таковым является описанный ранее оператор **Set**, который во входной инфиксной форме выглядит как =, например $x = a$, или $x = x + 1$. В отличие от многих других процедурных языков в „Математике“ не нужно заботиться об описании типа переменных, и одному и тому же символу можно присвоить целочисленные, вещественные, комплексные, символьные и т.д.

значения. Операции присвоения можно повторять несколько раз подряд, располагая их в одной ячейке и разделяя последовательные выражения знаком ; (точка с запятой):

x = 5; x = x + 1; x = x + 2
8

Разделенные точкой с запятой и идущие одно за другим выражения „Математики“ являются сами по себе выражением с заголовком **CompoundExpression**, который мы будем переводить как *составное выражение.*

FullForm[Hold[x = 5; x = x + 1; x = x + 2]]
Hold[*CompoundExpression*[*Set*[*x*, 5], *Set*[*x*, *Plus*[*x*, 1]]
Set[*x*, *Plus*[*x*, 2]]]]

Как в правой, так и в левой частях оператора присвоения могут стоять произвольные выражения „Математики“, поэтому нам не надо останавливаться на арифметических, логических и других операциях, поскольку они нами ранее использовались.

Оператором, в известном смысле имитирующим **CompoundExpression**, является оператор **Do**. Приведем его возможно более полное описание. Выражение **Do[expr, {imax}]** при его выполнении *imax* раз вычисляет выражение expr. Оно полностью эквивалентно выражению **expr; expr; ...; expr;**, в котором точка с запятой поставлена и после последнего выражения *expr*. Таким образом, в результате вычисления рассматриваемых выражений в выходную ячейку ничего не помещается, хотя *expr* вычислено.

t = 3; Do[t = t + 1/t, {3}]
t
$\frac{12781}{3270}$

Если желательно сразу вывести на экран значение *t*, то нужно ввести следующее составное выражение:

t = 3; Do[t = t + 1/t, {3}]; t

$\frac{12781}{3270}$

В рассматриваемом случае применение оператора **Do** полностью эквивалентно применению функции **Nest** в виде **Nest[(# + 1/#)&, 3, 3]**.

Выражение **Do[expr, {i, imax}]** при его выполнении вычисляет *imax* раз выражение *expr*, которое может содержать символ *i*. При этом *i* изменяется от 1 до *imax* с шагом единица.

t = 3; Do[t = t + i, {i, 3}]; t

9

Списки **{3}** или **{i, 3}** в операторе **Do** называются *итераторами*. Итератор может быть задан и в виде **{i, imin, imax, step}**, где *imin* — начальное значение *i*, *imax* — конечное значение, *step* — шаг изменения *i*. Если параметр *step* отсутствует, то он по умолчанию считается равным единице. Параметр *i* может быть любым выражением, заголовок которого не имеет атрибута *Protected*, а параметры *imin, imax* и *step* не обязательно числа, но выражение **(imax – imin)/step** должно быть числом. Кроме того, шаг *step* может быть отрицательным при условии, что *imax* меньше *imin*.

t = 3; Do[t = t + f[x], {f[x], y, 3.5y, y}]; t

$3 + 6y$

В операторе **Do** может быть несколько итераторов:

Do[Print[i a + j b], {i, 3}, {j, i}]

$a + b$

$2a + b$

$2a + 2b$

$3a + b$

$3a + 2b$

$3a + 3b$

Последний рассмотренный пример демонстрирует применение оператора **Do** для получения так называемых побочных эффектов. В самом деле функция **Print** не изменила значение ни одного выражения.

Функциями, тесно связанными с **Do**, являются **Sum**, **Product** и **Table**. Вместо составного выражения

i = 0; Do[i = i + j^k, {j, 5}]; i

дающего после вычисления следующий результат:

$1 + 2^k + 3^k + 4^k + 5^k$

можно ввести выражение с заголовком **Sum**:

Sum[j^k, {j, 5}]
$1 + 2^k + 3^k + 4^k + 5^k$

Аналогичные вычисления выполняет функция **Product**.

Product[j^k, {j, 5}]
$2^k 3^k 4^k 5^k$

7.2. Условные операторы

Условные операторы позволяют ветвить вычисления в зависимости от выполнения одного или нескольких условий, выраженных предикатами. Выражение

If[test, expr1, expr2]

имеет значением вычисленное выражение *expr1*, если *test* вычисляется на **True**, и вычисленное выражение *expr2*, если значение *test* равно **False**. Если же *test* не вычисляется ни на **True**, ни на **False**, то исходное выражение с заголовком **If** остается невычисленным.

{If[3 < 7, x, y], If[3 > 7, x, y], If[z < 7, x, y]}
$\{x, y, If[z < 7, x, y]\}$

Возможно использование функции **If** с двумя или четырьмя аргументами. Выражение **If[test, expr]** имеет значением *expr*, если вычисленный *test* равен **True**; значением *Null*, если *test* равен **False**, и остается невычисленным в противном случае. Если произошло вычисление на *Null*, то в выходную ячейку ничего не помещается, и она попросту отсутствует.

Рассмотрим в качестве примера следующую задачу. Пусть для заданного списка, элементами которого являются числа, требуется увеличить их значение на единицу, если элемент меньше 5, и оставить неизменными в противном случае. Эту задачу можно выполнить, например, с помощью следующей программы:

```
list1 = Table[i, {i, 10}];
Do[If[list1[[i]] < 5, list1[[i]] ++], {i, 10}]; list1
{2, 3, 4, 5, 5, 6, 7, 8, 9, 10}
```

В примере *list*1[[*i*]] + + означает *list*1[[*i*]] = *list*1[[*i*]] + 1.

Выражение **If[test, expr1, expr2, expr3]** при условии, что тест не принимает ни одного из двух булевых значений, имеет результатом вычисленное *expr3*. Используем этот вариант функции **If** для решения усложненной предыдущей задачи. Поставим условие, что если среди элементов списка встречаются нечисловые элементы, то вместо них должно быть поставлено число 0.

```
l2 = Append[list1, {z}]; Do[If[l2[[i]] < 5, l2[[i]] ++,
l2[[i]], l2[[i]] = 0], {i, 11}]; l2
{3, 4, 5, 5, 5, 6, 7, 8, 9, 10, 0}
```

Допускается вложенное вхождение функций **If**, как это демонстрируется в следующем примере:

```
f[x_] := If[x < 0, 0, If[x < 1, x, 1]]
```

Нами определена функция $f[x]$, равная нулю при $x < 0$, единице при $x > 1$ и равная x на отрезке от 0 до 1. Естественно, такое вложенное вхождение можно повторять как угодно большое

число раз, но в „Математике“ предусмотрена другая функция, эквивалентная вложенному вхождению **If**.

Условная функция

Which[test1,expr1,test2,expr2,...]

имеет четное число $2n$ аргументов, среди которых n тестов *testi* и n выражений *expri*. При этом вычисляется первое из *expri*, для которого стоящий перед ним *testi* равен **True**. Если первые k тестов равны **False**, а $k+1$-й имеет небулево значение, то вся функция с заголовком **Which** остается невычисленной.

{Which[3 > 7,x,5 > 7,y,9 > 7,z],
Which[3 > 7,x,a > 7,y,9 > 7,z]}
$\{z, Which[3 > 7, x, a > 7, y, 9 > 7, z]\}$

В случае, когда все тесты вычисляются на **False**, значение введенного выражения с заголовком **Which** равно *Null*. Иногда бывает удобно вместо результата *Null* помещать в выходную ячейку какое-то сообщение, позволяющее судить, что имеет место рассматриваемая ситуация с тестами. Для этого последним тестом делается **True**, а следующим за ним выражением делается сообщение.

Which[3 > 5,x,4 > 5,y,True,"Fail"]
Fail

В качестве примера применения функции **Which**, определим функцию, которая по координатам точки M на плоскости определяла бы, в каком из четырех квадрантов находится эта точка. Для простоты будем считать, что M не лежит на координатных осях.

quadrant[{x_,y_}] := Which[x > 0 && y > 0,1,
x < 0 && y > 0,2,x < 0 && y < 0,3,x > 0 && y < 0,4]
{quadrant[{-3,5}],quadrant[{2,-7}],quadrant[{-3,-1}],
quadrant[{1,5}]}
{2,4,3,1}

Последняя из рассматриваемых функций, функция **Switch**, строго говоря, не может быть названа условной, так как она приводит к ветвлению вычислений не в результате проверки выполнения какого-то условия-предиката, а проверяет соответствие выражения шаблонам. При вычислении выражения

Switch[expr, pattern1, expr1, pattern2, expr2, ...]

проверяется соответствие вычисленного выражения *expr* шаблонам *patterni*. Если *patternk* первый шаблон, которому оно соответствует, то результатом вычисления рассматриваемой функции будет вычисленное выражение *exprk*. При несоответствии ни одному шаблону, вычисление с заголовком **Switch** помещается в выходную ячейку невычисленным. Однако если последний шаблон равен _, то результатом будет вычисленное последнее выражение *exprn*.

Рассмотрим в качестве примера программу *norm*, которая нормализует списки по длине, превращая любой непустой список в список длины 5, в соответствии со следующим правилом. Если длина списка меньше пяти, то в его конец добавляются нули, если больше пяти, то лишние элементы в конце списка удаляются. Пустой список остается неизменным. Если на входе программы оказывается выражение, не являющееся списком, то выдается сообщение *input is not a list.*

```
SetAttributes[norm, HoldAll]
norm[l_] :=
    Switch[l, _List, Which[Length[l] == 0||
    Length[l] == 5, l, Length[l] < 5,
    Do[l = Append[l, 0], {5 - Length[l]}];
    l, Length[l] > 5, Do[l = Delete[l, -1], {Length[l] - 5}];
    l], _, "input is not a list"]
```

Мы снабдили функцию **norm** атрибутом *HoldAll* для того, чтобы ее аргумент не вычислялся раньше, чем будет использовано

данное определение. В противном случае в операторе **Do** происходило бы присваивание списку l другого списка, скажем, списка **Delete[l,–1]**. Создадим теперь три списка $l1, l2, l3$, а символу $l4$ присвоим значение выражения $x + y$.

{l1 = {}, l2 = {a, b, c}, l3 = {a, b, c, d, e, f, g}, l4 = x + y}
$\{\{\}, \{a, b, c\}, \{a, b, c, d, e, f, g\}, x + y\}$

{norm[l1], norm[l2], norm[l3], norm[l4]}
$\{\{\}, \{a, b, c, 0, 0, \}, \{a, b, c, d, e\},$ *input is not a list*}

Таким образом, поставленная нами задача решена.

7.3. Условные циклы

Цикл с заголовком **While** имеет вид

While[test, expr]

Его вычисление начинается с проверки условия *test*. Если оно имеет значением True, то вычисляется *expr*, которое при вычислении каким-либо образом изменяет *test*. Далее опять проверяется тест, и так происходит до тех пор, пока *test* не получит значения, отличного от True. По окончании цикла формальным результатом вычислений является *Null*. Таким образом, цикл **While** никогда не остается невычисленным. Проиллюстрируем сказанное простейшими примерами:

x = 1; While[x < 7, t = x²; x + +]; t
36

x = 10; While[x > 5, Print[x" > 5"]; x − −]
10 > 5
9 > 5
8 > 5
7 > 5
6 > 5

Более содержательный пример доставляет следующая программа, реализующая метод бисекций нахождения нулей функций. Метод состоит в следующем. Допустим, что мы нашли две точки a и b в окрестности нуля функции $f(x)$, такие, что в этих точках функция принимает значения противоположных знаков. Тогда отрезок $[a, b]$ делится пополам, и если знак функции f в середине отрезка совпадает с ее знаком в точке a, то середина отрезка принимается за новую точку a. Если же знак в середине отрезка совпадает со знаком функции в точке b, то середина отрезка принимается за новую точку b. Процесс деления продолжается, пока модули значений функции на концах отрезка не сделаются достаточно малы.

```
bisect[f_, a_, b_, eps_] := (If[N[f[a]f[b]] > 0,
z = "a, b are not good", t = a;  s = b;
While[Max[N[f[t]], N[f[s]]] > eps, u = N[f[t]];
v = N[f[s]]; w = N[f[(t + s)/2]];  If[uw >= 0, t = (t + s)/2];
If[vw > 0, s = (t + s)/2];  z = N[t]]];  z)
```

В программу включена проверка условия противоположности знаков функции на концах отрезка.

```
bisect[AiryAiPrime, 1, 2, 0.001]
```
a,b are not good

```
bisect[AiryAiPrime, -2, 2, 0.001]
```
−1.01953

Оператор условного цикла с заголовком **For** устроен следующим образом. Выражение

```
For[start, test, step, expr]
```

инициирует некоторые начальные данные *start*, затем вычисляет многократно выражения *step* и *expr*, пока условие *test*

вычисляется на True. Как и для цикла с заголовком **While**, формальный результат вычисления цикла **For** есть *Null*. Вот простой пример применения этого цикла.

For[i = 2, i < 5, i + +, Print[Expand[(1 + x)^i]]]

$1 + 2x + x^2$

$1 + 3x + 3x^2 + x^3$

$1 + 4x + 6x^2 + 4x^3 + x^4$

7.4. Функция Module

Интерактивный характер проведения сеансов работы с „Математикой" приводит к необходимости писать программы во время текущей работы, в момент, когда, возможно, уже не одному десятку символов присвоены определенные значения. В то же время при написании программ приходится использовать вспомогательные переменные, относительно которых пользователь молчаливо предполагает, что им не присвоены какие-либо значения. Если это предположение ошибочно, то программа может начать вести себя непредсказуемым образом. Даже после безошибочной работы программы в качестве побочного эффекта ее выполнения таким вспомогательным переменным оказываются присвоенными некоторые значения. Это явление носит название *конфликта символов*. Правда, переменные итераторов в операторе **Do**, скажем, переменная i в **t = 0; Do[t = t + i^2, {i, 4}]** локализована, поэтому после выполнения рассмотренного составного выражения символ i имеет то же значение, что и до его выполнения, но использованный в качестве вспомогательной переменой символ t получил значение 30. В операторе **For** переменная цикла не локализована, поэтому после выполнения программы

t = 0; For[i = 1, i < 5, i + +, t = t + i^2]

символ i имеет значение 5. В „Математике" имеется функция, которая позволяет пользователю локализовать вспомогательные выражения так, что, несмотря на совпадение вспомогательных переменных с символами, значения которых были ранее определены, это не приводит к нарушениям в работе программы. Аналогично, после выполнения программы такие символы имеют те же значения, что они имели раньше. Таким программным средством локализации является функция **Module**.

Функция **Module** имеет два аргумента. Первый есть список локальных переменных, второй — исполняемая программа:

Module[{x,y,...},expr]

Локальным переменным списка, всем или их части, можно присвоить начальные значения, т.е. задавать выражения с заголовком **Module** в виде

Module[{x = x0,y,...},expr]

Значением функции **Module** является вычисленное *expr*.

Механизмом локализации является создание временных вспомогательных переменных вида $x\$n$, где n есть значение системной переменной *$ModuleNumber*. Значение этой переменной увеличивается на единицу всякий раз, когда происходит вычисление любой функции с заголовком **Module**.

Table[Module[{t},t],{i,3}]
{*t*$1, *t*$2, *t*$3}

Символы вида $x\$n$ снабжаются атрибутом *Temporary*, который предписывает „Математике" удалять символы, которые не были экспортированы в выходные ячейки, после окончания вычисления функции **Module**.

Module[{t}, Print[t]]

$t\$4$

?t*

$t\$1, t\$2, t\$3$

Из сказанного выше следует, что при определении, например, функции **bisect** нам следовало бы воспользоваться функцией **Module**.

```
bisect[f_, a_, b_, eps_] := Module[{z, u, v, w, t = a, s = b},
If[N[f[a]f[b]] > 0, z="a,b are not good",
While[Max[N[f[t]], N[f[s]]] > eps, u = N[f[t]]; v = N[f[s]];
w = N[f[(t + s)/2]]; If[uw >= 0, t = (t + s)/2];
If[vw > 0, s = (t + s)/2]; z = N[t]; z]
```

Упражнения

1. С помощью функции **Do** *напишите* программу **fact[n _]** вычисления $n!$ для натуральных n. Предусмотрите, чтобы при аргументе, не являющемся натуральным числом, было напечатано соответствующее сообщение. В программе используйте функцию **Module**.

2. Числа Фибоначчи f_n определяются с помощью рекуррентного соотношения $f_n = f_{n-1} + f_{n-2}$. *Напишите* программу **fib[n _]** их вычисления с помощью функции **Do**. Предусмотрите те же проверки аргумента, что и в предыдущем упражнении. Сравните время вычисления **fib[100]** по написанной программе с временем программ в стиле правил преобразований и в стиле динамического программирования.

3. Встроенная функция **Prime[n]** имеет значением n-е по номеру простое число. Используя функцию **While**, *напишите* программу, определяющую число простых чисел, меньших или равных n.

4. *Напишите* программу **cnt[l_List]**, которая по заданному списку l порождает новый список, элементами которого являются пары. Первыми элементами пар служат все различные элементы списка l, вторыми — число их вхождений в l. Используйте функции **For**, **Do** и **Module**.

В следующих задачах не требуется, чтобы обязательно использовался процедурный стиль программирования. Единственное требование: наличие функции Module.

5. *Определите* функцию changeto, первым аргументом которой является натуральное число n, вторым аргументом – список натуральных чисел. В результате вычисления этой функции должен быть создан список способов размена суммы в n рублей на купюры достоинством, равным числам из второго списка.

6. (Латинские квадраты). Латинским квадратом называется квадратная $n \times n$ матрица, элементами которой являются натуральные числа от 1 до n и удовлетворяющая условию: ни в одной строке и ни в одном столбце не должно быть повторяющихся чисел. *Определите* функцию latinsquare[n_Integer?Positive], которая порождала бы случайным образом латинский $n \times n$ квадрат.

7. Два латинских квадрата P и Q называются ортогональными, если среди n^2 пар (P_{ij}, Q_{ij}) матричных элементов нет одинаковых. *Напишите* программу orthQ проверки латинских квадратов на ортогональность. *Найдите*, используя функцию latinsquare, ортогональные квадраты четвертого и пятого порядков (в шестом порядке таковых нет).

Глава 8
ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЙ

Мы уже упоминали, что важнейшими понятиями, относящимися к компьютерной алгебре „Математика“, являются *выражение* и *вычисление.* Выражения подробно рассматривались в главе 4, наступило время познакомиться с принципами вычисления выражений. Самым существенным обстоятельством, которое здесь следует принять во внимание, является то, что рассматриваемая компьютерная алгебра есть система, вычислительный процесс в которой основан на правилах преобразований. Это означает, что при вычислении выражений последовательно применяются правила преобразований до тех пор, пока выражение не перестанет изменяться. Отсюда следует, что если вычисленное выражение, помещенное в выходную ячейку, вновь ввести во входную, то результат будет совпадать с введенным выражением. Более того, с введенным вычисленным выражением не будет происходить вообще никакого вычисления, в чем можно убедиться с помощью функции **Trace**:

```
Sin[x + 1] // Trace
{{x + 1, 1 + x}, Sin[1 + x]}
Sin[1 + x] // Trace
{}
```

Такой результат объясняется тем, что каждое выражение „Математики“, за исключением чисел и строк, снабжается двумя метками, одна из которых несет информацию о времени вычисления выражений. Если эта метка не пуста, то выражение попросту не вычисляется. Вторая метка несет информацию

о наличии „ссылок“ на выражение. Например, выражение может быть присвоено в качестве значения какому-то другому выражению, и это другое выражение является ссылкой. Глобальное выражение *Out*[*n*] также является ссылкой. Выражение удаляется из памяти, если на него отсутствуют ссылки. Итак, вычисленное однажды выражение больше не вычисляется, т.е. не изменяется. В этом смысле не вычисляются также числа и строки. Таким образом, процесс вычислений существенно опирается на вычисление символов.

8.1. Значения, ассоциированные с символами

Напомним, что с каждым символом могут быть связаны следующие четыре типа правил преобразований. Если символу присвоено некоторое выражение с помощью функции **Set**, то это выражение является одним из *собственных значений* (**OwnValues**) символа.

```
x = f[y, z]
f[y, z]
OwnValues[x]
{Literal[x] :> f[y, z]}
```

Смысл функции **Literal** будет объяснен позже, а сейчас мы обратимся к другим правилам. Если символ является заголовком выражения, то с ним ассоциируются правила преобразований, которые называются его *нижними значениями* (**DownValues**).

```
g[x_, y_] := x + y
DownValues[g]
{Literal[g[x_, y_]] :> x + y}
```

Правила преобразований, которые определяют *верхние значения* (**UpValues**) символа, ассоциируются с символом, если он находится на первом уровне некоторого выражения.

x /: {x,y_,z_} = {x y,x z}
UpValues[x]
$\{Literal[\{x, y_, z_\}] :> \{x\,y, x\,z\}\}$

{x,a,b}
$\{a\,x, b\,x\}$

Если символ является заголовком заголовка некоторого выражения, то с ним могут быть ассоциированы его *дальние значения* (**SubValues**).

h[x_][y__] := x[y]
h[v][a,b]
$v[a, b]$

SubValues[h]
$\{Literal[h[x_][y__] :> x[y]]\}$

8.2. Атрибуты

Кроме правил преобразований, ассоциированных с символом *symb*, последний может быть снабжен некоторыми атрибутами, наделяющими функцию с заголовком *symb* определенными свойствами. С атрибутами *Protected* и *Listable*, которыми обладали некоторые встроенные функции, читатель уже знаком. Ниже мы более подробно остановимся на этих и других атрибутах и способе присвоения атрибутов символу. Ранее в гл. 5 нами была определена функция **cube** соотношением

cube[x_] := x^3

Если мы вычислим функцию **cube** от списка, то получим следующий интересный результат:

cube[{a,b,c}]
$\{a^3, b^3, c^3\}$

Функция **cube** ведет себя так, как если бы она обладала атрибутом *Listable*. Однако ясно, что этот атрибут на самом деле принадлежит функции **Power**, участвующей в определении функции **cube**. Чтобы избежать этого эффекта, определим две функции **concat** и **concatlist** с помощью встроенной функции **Join**, не обладающей атрибутом *Listable*.

```
concat[x_,y_] := Join[{x},{y}]
concat[{a,b},{2,3}]
{{a,b},{2,3}}
```

Если обратить внимание на фигурные скобки аргументов функции **Join**, то это довольно очевидный результат. Функцию **concatlist** мы зададим с помощью того же определения, что и функцию **concat**, но присвоим ей атрибут *Listable*. Отметим, что присвоение атрибутов функции должно предшествовать ее определению. Оно производится с помощью функции **SetAttributes**. А именно выражение **SetAttributes[symb,attr]** при его вычислении добавляет атрибуты *attr* к списку атрибутов символа *symb*.

```
SetAttributes[concatlist,Listable]
concatlist[x_,y_] := Join[{x},{y}]
concatlist[{a,b},{2,3}]
{{a,2},{b,3}}
```

Мы видим, что результаты вычислений обеих функций отличаются. Дело обстоит таким образом, что, прежде чем произвести преобразование введенного последнего выражения с помощью ассоциированного с символом **concatlist** определения, производится группировка в пары элементов списков, стоящих на одинаковых позициях, как это обычно происходит с аргументами функций, обладающих атрибутом *Listable*:

```
Power[{a,b},{2,3}]
```
$\{a^2, b^3\}$

Действие атрибута *Listable* таково же, как и действие функции **Thread**.

Thread[concat[{a, b}, {2, 3}]]
{{*a*, 2}, {*b*, 3}}

Удаление атрибута производится с помощью функции **ClearAttributes**.

ClearAttributes[concatlist, Listable]
concatlist[{a, b}, {2, 3}]
{{*a*, *b*}, {2, 3}}

Теперь результаты вычислений функций **concat** и **concatlist** совпадают.

Функции **Map**, **MapAt** и **Thread** позволяют добиваться того же результата, что и присвоение атрибута *Listable*.

{f[{a, b, c}], Map[f, {a, b, c}]}
{*f*[{*a*, *b*, *c*}], {*f*[*a*], *f*[*b*], *f*[*c*]}}
{f[{a, {b, c}}], MapAt[f, {a, {b, c}, {{1}, {2, 1}, {2, 2}}}]}
{*f*[{*a*, {*b*, *c*}}], {*f*[*a*], {*f*[*b*], *f*[*c*]}}}

Всего имеется четырнадцать атрибутов функций, но мы остановимся только на некоторых.

Атрибут *Orderless* означает, что наделенная им функция безразлична к порядку ее аргументов подобно встроенным функциям **Plus** и **Times**.

SetAttributes[f, Orderless]
f[y, 1, x]
f[1, *x*, *y*]

Аналогичного результата можно добиться с помощью функции **Sort**.

ClearAttributes[f, Orderless]
Sort[f[y, 1, x]]
f[1, *x*, *y*]

Атрибут *Flat* соответствует свойству ассоциативности функций.

SetAttributes[g, Flat]
g[a, g[b, c]]
g[*a*, *b*, *c*]

Этим атрибутом наделены встроенные функции **Plus** и **Times** в соответствии с тем, что операции сложения и умножения ассоциативны: $(a+b)+c=a+(b+c)$ и $(ab)c=a(bc)$.

(a b) c // FullForm
Times[*a*, *b*, *c*]

Попробуем удалить атрибут *Flat* из списка атрибутов функции **Times**.

ClearAttributes[Times, Flat]
(a b) c // FullForm
Times[*Times*[*a*, *b*], *c*]

Атрибут *Flat* действует так, как если бы к результату вычисления не обладающей этим атрибутом функции была бы применена функция **Flatten**.

h[h[a, b], h[c]]
h[*h*[*a*, *b*], *h*[*c*]]

h[h[a, b], h[c]] // Flatten
h[*a*, *b*, *c*]

Атрибут *OneIdentity*, в отличие от ранее рассмотренных атрибутов, воздействует только на процесс сопоставления выражений шаблонам. Он означает, что с точки зрения соответствия шаблонам выражения **f[x]**, **f[f[x]]** и т.д. для функций **f**, *обладающих атрибутом Flat*, эквивалентны **x**. Рассмотрим несколько примеров, разъясняющих суть дела. Пусть сначала функция **f** не обладает никакими атрибутами. Вычислим выражение:

f[f[x]]/.f[z_] :> z^2
$f[x]^2$

В рассматриваемом случае „Математика“ отождествила заголовок шаблона с заголовком выражения **f[f[x]]**, а шаблон *z_* с подвыражением **f[x]**. Присвоим функции **f** атрибут *Flat*

SetAttributes[f, Flat]

и вновь вычислим рассматриваемое выражение.

f[f[x]]/.f[z_] :> z^2
$f[x]^2$

Результат не изменился, так как при сопоставлении с шаблонами учитывается не только вычисленное выражение **f[f[x]]**, но и его исходная форма. Допустим, что функция **f** обладает и атрибутом *Flat*, и атрибутом *OneIdentity*.

SetAttributes[f, {Flat, OneIdentity}]
f[f[x]] /. f[z_] :> z^2
x^2

Мы видим, что результат таков, как если бы функция **f** была аналогична, скажем, функции **Plus** в том смысле, что ее применение к единственному аргументу ничего не изменяет: **Plus[5]** после вычисления дает 5.

Атрибут *Constant* присваивается символам, которые должны вести себя как константы по отношению к операторам дифференцирования.

```
Attributes[f] = .
D[f[x],x]
f'[x]
SetAttributes[f, Constant]
D[f[x],x]
0
```

Подобно тому как атрибут *Protected* препятствует присвоению нижних значений (**DownValues**) символам, атрибут *Locked* не позволяет изменять атрибуты символов.

```
SetAttributes[f, Locked]
Attributes[f]
{Constant, Locked}
ClearAttributes[f, {Constant, Locked}]
Attributes:: locked: symbol f is locked
```

Придется начать новую сессию „Математики“, если возникнет необходимость изменить атрибуты символа **f**. Будем считать, что мы это сделали, поскольку мы будем использовать ниже символ **f**. Если функции какого-либо пакета снабжены атрибутами *Protected* и *Locked*, то пользователь не сможет изменить функций, определенных в этом пакете, если он подгружен.

Атрибут *ReadProtected* не позволяет прочитать определений, ассоциированных с символом, с помощью функции **Information**.

```
f = .; f[x_] := x^3
?f
Global`f
f[x_] := x^3
```

SetAttributes[f,ReadProtected]
?f
Global'f
Atributes[f] = {ReadProtected}

Если символ снабжен атрибутами *ReadProtected* и *Locked*, то в текущей сессии нельзя узнать ассоциированных с ним определений.

8.3. Стандартный процесс вычислений

При вычислении выражений „Математика" руководствуется следующими принципами, некоторые из которых были рассмотрены в начале этой главы.

1. Вычисленные ранее выражения, а также строки, числа и символы, не имеющие собственных значений (**OwnValues**), не изменяются.

2. Порядок и ход вычислений контролируются заголовками выражений и подвыражений.

3. *Стандартный порядок вычислений* означает, что сначала вычисляется заголовок выражения, потом непосредственные подвыражения последовательно слева направо.

4. На самых ранних стадиях вычислений символы заменяются на присвоенные им собственные значения.

5. Если в процессе вычисления ни одна часть выражения не изменилась, вычисления прекращаются.

6. После того как вычислен заголовок выражения (или заголовки подвыражений), выполняются последовательно преобразования, индуцированные атрибутами *Flat*, *Listable* и *Orderless*.

7. Выполняются заданные пользователем преобразования, составляющие верхние значения (**UpValues**) символов-заголовков. Причем в первую очередь выполняются менее общие, а затем более общие определения.

8. Выполняются преобразования, являющиеся встроенными верхними значениями заголовков, в том же порядке, что и в п.7.

9. Применяются правила преобразований, определенные пользователем и составляющие нижние значения (**DownValues**) заголовков-символов.

10. Если заголовок — не символ, то используются правила преобразований, связанные с заданными пользователем дальними значениями (**SubValues**) символов, которые являются заголовками заголовков.

11. Используются встроенные правила преобразований, являющиеся нижними значениями вычисленных символов-заголовков. Замечания об общности правил те же, что и раньше.

Проиллюстрируем сказанное простейшими примерами:

b = 2; x^3 + b x + b^2 // Trace

$\{\{\{b,2\},2x\},\{\{b,2\},2^2,4\},x^3+2x+4,4+2x+x^3\}$

Заголовок *Plus* вычисляемого выражения и символ x не имеют **OwnValues**, поэтому, так же как и выражение **x^3**, они не преобразуются. Поэтому мы не видим следов процесса их вычисления. Фактические вычисления начинаются с элемента $b\,x$, затем вычисляется b^2. После этого выполняется сортировка выражения, так как функция **Plus** обладает атрибутом *Orderless*.

f = Times; f[2, x + 3] // Trace

$\{\{f,Times\},\ \{x+3,3+x\},\ \{2(3+x)\}\}$

Здесь мы видим, что заголовок выражения вычисляется первым, число 2 не преобразуется, выражение **x + 3**, имеющее заголовок **Plus**, подвергается сортировке, а затем элементы вычисляемого выражения соединяются в произведение.

f = .; f[x_, y_] := x^y; f[3 + 2, b + a] // Trace

$\{\{3+2, 2+3, 5\}, \{b+a, a+b\}, f[5, a+b],, 5^{(a+b)}\}$

Этот пример иллюстрирует, что правила преобразования применяются после вычисления аргументов. Оставим в силе определение, ассоциированное с f, и добавим новое, являющееся верхним значением для символа g:

g/ : f[g[x_], y_] := g[x + y]
f[g[x + 2], 3 + y] // Trace

$\{\{\{x+2, 2+x\}, g[2+x]\}, f[g[2+x], 3+y], g[2+x+(3+y)],$
$\{2+x+(3+y), 2+x+3+y, 2+3+x+y, 5+x+y, g[5+x+y]\}\}$

ClearAll[f, g]

Здесь хорошо видно, что верхние значения используются раньше, чем нижние.

8.4. Выражения, вычисляемые нестандартно

Во многих ситуациях, с некоторыми из которых мы познакомимся ниже, бывает необходимо изменить стандартный порядок вычисления выражений. Применение самого крайнего средства: вообще не вычислять введенного выражения — обеспечивает функция **Hold**.

f[1 + 1, x + 2]

$f[2, 2+x]$

Hold[f[1 + 1, x + 2]]

$Hold[f[1+1, x+2]]$

Функция **HoldForm** аналогична по действию функции **Hold**, но в отличие от последней она не воспроизводится в выходной ячейке.

HoldForm[f[1 + 1, x + 2]]
$f[1+1,x+2]$

Функция **ReleaseHold** устраняет заголовки *Hold* и *HoldForm*, поэтому после ее применения выражение — аргумент последних функций вычисляется.

ReleaseHold[%]
$f[2,2+x]$

Общим механизмом, позволяющим изменить стандартный процесс вычислений, является наличие специальных атрибутов *HoldAll*, *HoldFirst* и *HoldRest*. Эти атрибуты блокируют вычисление соответственно всех непосредственных подвыражений, первого непосредственного подвыражения или всех непосредственных подвыражений, кроме первого у выражения, заголовок которого имеет рассматриваемые атрибуты. Функции **Hold** и **HoldForm** имеют атрибут *HoldAll*.

Attributes[f] = HoldFirst
f[1 + 1, x + 2]
$f[1+1,2+x]$

Атрибут *HoldFirst* имеют многие встроенные функции, например **Set**.

Attributes[Set]
$\{HoldFirst, Protected\}$

f := a; g := b
f = g // Trace
$\{\{g,b\},f=b,b\}$

?f
Global`f
$f = b$

При вычислении выражения $\mathbf{f = g}$ было использовано правило преобразования, ассоциированное с символом g, но не с f. Что было бы, если бы функция **Set** не имела атрибута *HoldFirst*?

Было бы ошибкой думать, что первый аргумент у функции, имеющей атрибут *HoldFirst*, вообще никогда не вычисляется. Он не вычисляется на первых этапах стандартного процесса вычислений, а именно на этапах, предшествующих применению правил преобразований, ассоциированных с заголовками. Если же при применении правил преобразований рассматриваемый первый аргумент стал аргументом функции с заголовком, не имеющим атрибутов *Hold**, или не попал под действие таких атрибутов, то он вычислится. Таким образом, рассматриваемые атрибуты позволяют передавать невычисленные аргументы в выражения с другими заголовками.

SetAttributes[f, HoldFirst]
f[x_, y_Integer] := (x = Delete[x, y])
x = {a, b, c}; f[x, 2]
$\{a, c\}$
x
$\{a, c\}$

Другая задача, решаемая с помощью атрибутов *Hold** — сделать более гибким применение шаблонов.

SetAttributes[g, HoldFirst]
g[a + 2, b] /. g[x_ + y_, z_] :> (x + z)y
$2(a + b)$

Кроме функций **Set**, имеющей атрибут *HoldFirst*, и **SetDelayed** с атрибутом *HoldAll* большинство функций процедурного программирования и графических функций имеют атрибуты *Hold**.

Attributes /@ {If, Do, For, While, Plot, Show}
{{*HoldAll*, *Protected*}, {*HoldAll*, *Protected*},
{*HoldAll*, *Protected*}, {*HoldAll*, *Protected*},
{*HoldAll*, *Protected*}, {*Protected*}}

Это позволяет таким функциям вычислять все или некоторые свои аргументы нестандартно. Так, функция **Do** вычисляет свой первый аргумент несколько раз, функция **If** в зависимости от значения первого аргумента вычисляет либо второй, либо третий аргумент, функция **Plot** вычисляет свой первый аргумент несколько раз в некоторой последовательности точек и т.п.

Преодолеть воздействие атрибутов *Hold** на вычисление какого-то подвыражения e_k можно, если задать его в виде **Evaluate**[$\mathbf{e_k}$]. Например, функция **Plot** на раннем этапе своего вычисления опознает, является ли ее первый аргумент списком. Если является, то **Plot** делает необходимые приготовления для вычерчивания графиков сразу нескольких функций одновременно. Поэтому если первый аргумент является значением функции **Table**, то применение **Evaluate** необходимо, так как в противном случае графики не будут вычерчены. Функция **Evaluate** в некоторых случаях ускоряет вычисления. Так, вычисление выражения **ParametricPlot[{x[t], y[t]} /. n, {t, 0, Pi}]**, где *n* есть совокупность интерполяционных функций, хотя и возможно, но будет значительно ускорено и выполнено с большей точностью, если первый аргумент будет введен в виде **Evaluate[{x[t], y[t]} /. n]**.

В заключение этого параграфа отметим, что функциями, связанными с **Hold**, являются **HeldPart** и **ReplaceHeldPart**. Они позволяют выполнить следующие действия. Напомним,

что после извлечения функцией **Part** соответствующего заданной спецификации подвыражения, последнее вычисляется, даже если в исходном выражении оно находилось под действием атрибутов *Hold** и не вычислялось.

{g[1 + 1, y], Part[g[1 + 1, y], 1]}
{*g*[1 + 1, *y*], 2}

Вычисления извлеченной части можно избежать, если вместо функции **Part** применить функцию **HeldPart**.

HeldPart[g[1 + 1, y], 1]
Hold[1 + 1]

Мы видим, что **HeldPart** делает извлеченное подвыражение аргументом функции **Hold**.

При замене подвыражений, находящихся в области действия одного из атрибутов *Hold**, иногда возникает необходимость заменить часть таких подвыражений на новые без вычисления последних. Это осуществляется при помощи функций **Replace-HeldPart** и **Hold**.

{ReplaceHeldPart[g[x + y, v], Hold[a + 1], {1, 1}],
ReplacePart[g[x + y, v], Hold[a + 1], {1, 1}]}
{*g*[*a* + 1 + *y*, *v*], *g*[*Hold*[*a* + 1] + *y*, *v*]}

Рассматриваемая функция, по сравнению с функцией **ReplacePart**, дополнительно освобождает подставляемое выражение от заголовка **Hold**.

8.5. Вычисление правил преобразований

Преобразования в соответствии с шаблонами и использование шаблонов в локальных и глобальных правилах преобразований являются одной из самых сильных сторон „Математики" как

системы символьных преобразований. Естественно поэтому, что при вычислении выражений, включающих шаблоны, встречается ряд нестандартных моментов. Общий принцип вычисления выражений с шаблонами следует из того простого факта, что применение шаблонов начинается на таких этапах вычислений, когда выражения, к которым применяются преобразования по шаблонам, уже частично вычислены. Следовательно, и правила преобразований при этом хотя бы частично, но вычисляются.

Проследим за ходом простейших вычислений с шаблонами. Начнем с локальных правил преобразований:

x = 2; y = 3; h[x^2 + y] /. h[2x + y] → 1/x

$$\frac{1}{2}$$

Преобразование выражения **h[7]** в 1/2 произошло потому, что левая часть подстановки, т.е. выражение **h[2x + y]**, так же как и правая $1/x$, была вычислена прежде, чем выяснение соответствия шаблону. Если в правиле локальной подстановки использовать **RuleDelayed**, то получится тот же результат:

h[x^2 + y] /. h[2x + y] :> 1/x

$$\frac{1}{2}$$

Подстановка **h[x_] :> 1/x** также выполнится, но даст результат 1/7.

h[x^2 + y] /. h[x_] :> 1/x // Trace

$\{\{\{\{\{x,2\},2^2,4\},\{y,3\},4+3,3+4,7\},h[7]\},$

$h[7]\ /.\ h[x_] :> \frac{1}{x},\frac{1}{7},\{\frac{1}{7},\frac{1}{7}\},\frac{1}{7},\frac{1}{7},\frac{1}{7}\}$

Итак, резюмируя, можно сказать: в локальных правилах преобразований вида *lhs* → *rhs* и *lhs* :> *rhs* происходит вычисление

левой части до установления соответствия шаблону при условии, что *lhs* не содержит шаблон _.

Иначе обстоит дело в глобальных правилах преобразований. Мы знаем, что функция **Set** имеет атрибут *HoldFirst*. Именно поэтому для символов в левой части определения $x = expr$ не используются их **OwnValues**, и эти значения тем самым переопределяются. Однако, несмотря на наличие этого атрибута, в выражениях вида $f[args] = expr$ аргументы *args* вычисляются.

```
x = 2; y = 3; h[x^2 + y] = z
z
?h
Global`h
h[7] = z
```

Исключениями из этого правила являются функции с атрибутами *Hold**. Кроме того, если желательно избежать вычисления *args*, то к нему можно применить функцию **Literal**, играющую ту же роль, что и **Hold**. Разницу в применениях **Literal** и **Hold** можно понять из следующего примера.

```
Clear[x,y,g,h]; h[Literal[x^2 + y]] = z;
g[Hold[x^2 + y]] = w;
h[x^2 + y]
z
{g[x^2 + y], g[Hold[x^2 + y]]}
{g[x^2 + y], w}
```

Рассмотрим еще один пример использования функций **Hold** и **Literal**. Пусть символу x присвоено значение 2. Как преобразовать выражение **x^3** в **z^3**? Это можно сделать следующим образом.

```
x = 2; Hold[x^3] /. Literal[x] -> z // ReleaseHold
```
z^3

Упражнения

1. *Ответьте* только на основе анализа принципов вычисления выражений, будет ли получен список $\{1,2,3\}$, если вычислить выражение **Apply[List, 1 + 2 + 3]**? Будет ли получен список $\{a,b,c\}$, если в этом выражении заменить $1+2+3$ на $a+b+c$? Как из выражения $1+2+3$ получить список $\{1,2,3\}$, применяя функции **Hold, ReleaseHold, Apply**?

2. Допустим, что мы хотим определить функцию f, такую, что она любую сумму слагаемых преобразует в список слагаемых. Будет ли наша цель достигнута, если мы дадим определение

   f[x_Plus] := Apply[List, Hold[x], 2] // ReleaseHold?

   Как изменить это определение, чтобы все было в порядке?

3. Выражение **Information[symb]**, или **?symb**, дает справку о символе *symb*. *Как изменить* атрибуты функции **Information**, чтобы получить справку о каждом символе f, g и т.д. из списка $\{f,g,\ldots\}$ одновременно?

4. Ассоциируйте с символом h определение

   h[x _Symbol, y _Symbol] := {x, y}

   и снабдите h атрибутом Flat. *Предскажите*, не вычисляя заранее, каков будет результат вычисления выражения **h[a, b, c, d]**? Каков будет результат, если дополнительно снабдить h атрибутом OneIdentity? *Посмотрите*, используя функцию **Trace**, как „Математика" вычисляет это выражение в первом и втором случаях.

5. Вычисление выражения **x = N[x]** дает результат x. В то же время, определив функцию

   f[x_] := (x = N[x]; Sqrt[x])

   и вычисляя, скажем **f[3]**, мы получаем сообщение об ошибке и результат *Sqrt*[3]. Почему? *Как изменить* определение, чтобы функция f от аргумента типа Integer имела результатом вещественное число?

Глава 9
РАЗРАБОТКА ПРОГРАММ

Материал предшествующих глав позволяет читателю вести замкнутое „натуральное хозяйство“ в пределах одной отдельно взятой Записной книжки, т.е. проводить вычисления и графические построения, пользуясь встроенными функциями, функциями из подгружаемых стандартных пакетов и самостоятельно определенными функциями. Если последних становится много или они представляют собой многострочные совокупности определений, то возникает необходимость их сохранять и вызывать точно так же, как вызываются функции пакетов, являющихся неотъемлемой частью „Математики“. Основное требование, которому должны удовлетворять такие пакеты — отсутствие конфликта импортируемых функций, констант и переменных с введенными ранее в текущей сессии. Это требование становится еще более существенным, если иметь в виду дальнейший обмен программами с коллегами или их коммерческое распространение. Совершенно невозможно предвидеть, какие определения будущий пользователь вашей программы успел связать, скажем, с символом f, являющимся в вашей программе заголовком основной или вспомогательной функции. Отсутствие подобных конфликтов при подгрузке стандартных пакетов позволяет предположить, что разработчики „Математики“ нашли или использовали эффективный метод решения рассматриваемой задачи.

9.1. Контексты

Если пользователь однажды ввел с клавиатуры какой-либо символ, то „Математика“ будет рассматривать его как определен-

ный „глобально", т.е. как относящийся всюду в дальнейшем к одному и тому же объекту. Это позволяет связывать с символом различные определения, являющиеся основным механизмом символьных преобразований. Однако такое положение дел не совсем удобно, так как сколько-нибудь длительная работа требовала бы изощренной изобретательности в придумывании различных символов. Существует также определенная математическая и программистская традиция, следуя которой переменные итераторов обозначают i, j и т.п. Кроме того, часто возникает необходимость вводить вспомогательные переменные для обозначения промежуточных результатов. Все это объясняет существование механизмов локализации символов в определенных границах. Простейший такой механизм — механизм локализации *значений* итераторов в функциях **Do**, **Sum** и др.

```
i = 5; Sum[x^i, {i, 1, 3}]
```
$x + x^2 + x^3$
```
i
```
5

Несмотря на то что при вычислении суммы переменной i итератора в последний раз было присвоено значение 3, символ i сохраняет ранее присвоенное значение 5. Механизм локализации значений переменных в итераторах не препятствует конфликту символов. Рассмотрим простейший пример такого конфликта. Определим функцию, вычисляющую символьную сумму первых трех натуральных степеней своего аргумента.

```
f[x_] := Sum[x^i, {i, 3}]
{f[y], f[i]}
```
$\{y + y^2 + y^3,\ 32\}$

Второй результат довольно неожиданный, но если принять во внимание, что в этом случае функция **Sum** вычисляет сумму выражений вида $i\hat{}i$, он становится понятным.

Избежать подобного „захватывания“ итератором глобального символа позволяет функция **Module**. В данном примере определение

Clear[f]; f[x_] := Module[{i}, Sum[x^i, {i, 3}]]

позволяет избежать захватывания:

{f[y], f[i]}
$\{y + y^2 + y^3,\ i + i^2 + i^3\}$

Однако **Module** не препятствует конфликту символов-заголовков, определяемых с его помощью функций. Предположим, что мы забыли только что данное определение функции **f** и решили дать новое:

f[y_] := Module[{i}, Sum[y^i, {i, 4}]]
f[u]
$u + u^2 + u^3$

Мы смогли увидеть, что работает старое определение. Но как быть, если новое есть определение промежуточной функции, спрятанное среди других объектов в какой-то программе?

Механизм *контекстов* есть более совершенный механизм, позволяющий существенно ограничить конфликт символов. С чисто технической точки зрения контекст — это приставка к имени символа вида *Something*‘, где *Something* является именем контекста. *Любой символ всегда имеет непустую контекстную приставку.* Правда, как правило, она остается „за кадром“, не участвуя в представлении символа на экране дисплея. Следовательно, могут существовать разные символы, одинаково представляемые на экране. В этом смысле контекст играет роль директории (папки): ведь разные файлы с одним и тем же именем вполне могут сосуществовать, находясь в разных директориях. Узнать контекстную приставку символа

можно с помощью функции **Context**. Например, все встроенные функции и константы имеют контекстную приставку *System‘*.

Context /@{Sin, Pi, $MachinePrecision}
{*System‘*, *System‘*, *System‘*}

Текущее значение контекстной приставки, которая присоединяется к имени символа, можно узнать с помощью глобальной переменной *$Context* или с помощью рассмотренной ранее функции **Context**[ ], не указывая ее аргумента.

{$Context, Context[]}
{*Global‘*, *Global‘*}

По умолчанию, в начале сессии и до тех пор, пока не производились определенные, описываемые ниже манипуляции с контекстом, текущим контекстом является *Global‘*. Имя символа без контекстной приставки называется его *кратким* именем. При впечатывании с клавиатуры пользователь может присоединить любую приставку к символу.

{a‘x, a‘z, Global‘x}
{*a‘x*, *a‘z*, *x*}

В рассматриваемом случае контекстная приставка отображается на экране для символов *a‘x* и *a‘z* и не отображается для символа *Global‘x*. Чтобы разобраться в причинах этого, обратим внимание на два обстоятельства. Первое — текущий контекст есть Global. Второе — в настоящий момент контекста *a‘* нет на так называемой *контекстной дорожке*, содержание которой можно увидеть с помощью глобальной переменной *$ContextPath*.

$ContextPath
{*Global‘*, *System‘*}

Поместим контекст *a‘* на контекстную дорожку.

$ContextPath = Prepend[$ContextPath,"a‘"]
{*a‘*, *Global‘*, *System‘*}

{a‘x, a‘z, Global‘x}
{*a‘x*, *z*, *x*}

Разница в представлении символов *a‘z* и *a‘x* объясняется тем, что фигурировавший ранее в наших вычислениях символ *x* получил контекстную приставку *Global‘*, и, следовательно, существуют два символа с кратким именем *x*. Поскольку текущий контекст есть *Global‘*, то без приставки отображается символ *Global‘x*. До тех пор пока символ *a‘z* есть единственный символ с кратким именем *z*, к нему можно обращаться по его краткому имени даже в „чужом" для него текущем контексте *Global‘*.

z = 5;
{z, Context[z]}
{5, *a‘*}

Изменить текущий контекст на контекст *a‘* можно двумя способами: либо присвоить глобальной переменной $Context значение "*a‘*", либо прибегнуть к функции **Begin**:

Begin["a‘"];

После выполнения этой команды без контекстной приставки будет представлен символ *a‘x*.

{a‘x, a‘z, Global‘x}
{*x*, 5, *Global‘x*}

Можно сформулировать следующее общее правило, выполняющееся, когда вводится какой-либо символ без контекстной приставки. Если в текущем контексте есть полный символ с

кратким именем, совпадающим с вводимым символом, то вводимый символ интерпретируется как краткий символ текущего контекста. Если в текущем контексте соответствующего полного символа нет, то проверяется, есть ли полный символ в каком-либо контексте на контекстной дорожке. При наличии таких символов вводимому символу приписывается самый левый контекст из всех содержащих полный символ контекстов. Если на контекстной дорожке подходящих полных символов нет, то к введенному символу прибавляется контекстная приставка, совпадающая с именем текущего контекста.

v = 6; Context[v]
a‘

Предположим теперь, что определен новый контекст и присвоено некоторое значение ранее не встречавшемуся символу.

$Context = "b‘";
w = 13;

В текущем контексте *b‘* можно получить полную информацию о символе с кратким именем *w*.

?w
b‘w
w = 13

Вернемся, однако, к прежнему контексту и попытаемся получить информацию о символе *b‘w* по его краткому имени.

$Context = "a‘";
?w
Information:: notfound: Symbol w not found

Пример показывает, что если текущий контекст не совпадает с контекстной приставкой символа, то информацию о символе

по его краткому имени можно получить только в том случае, когда его контекст помещен на контекстную дорожку.

Существование нескольких полных символов с одним и тем же кратким порождает явление так называемого „затенения“ одного полного символа другим: символ с контекстной приставкой, расположенной на контекстной дорожке левее приставки символа с тем же кратким именем, затеняет последний. „Математика“ всегда предупреждает о возможности затенения.

Global‘z
Global‘z:: shdw:
Warning: Symbol z appears in multiple contexts {Global‘,a‘}; definitions in context Global‘ may shadow or be shadowed by other definitions

9.2. Контексты и программы

Механизм контекстов используется для того, чтобы хранящиеся в отдельном файле написанные пользователем определения основных и вспомогательных функций, т.е. разработанные пользователем программы, вводить в Записные книжки. Файл с такой программой в Версии 2.x должен иметь расширение *m*. Основные функции, ради которых и была написана программа, называются *экспортируемыми*. Программа начинается с команды

BeginPackage["NewContext‘"]

где *NewContext‘* — контекстная приставка, которой будут снабжены экспортируемые функции. Как правило, название вводимого контекста должно быть связано с областью приложения экспортируемых функций и быть по возможности уникальным, т.е. не совпадать ни с одним из известных. Хорошим примером могут служит стандартные пакеты „Математики“.

В них все графические функции имеют контекст, начинающийся с *Graphics‘*, функции для выполнения операций математического анализа имеют контекстную приставку, начинающуюся с *Calculus‘*, и т.д. Кроме того, возможно употребление подконтекстов: *Calculus‘Pade‘*. Команда **BeginPackage** изменяет контекстную дорожку и устанавливает текущий контекст равным *NewContext‘*. После ее выполнения контекстная дорожка, какова бы она ни была ранее, принимает вид:

{*NewContext‘*, *System‘*}

Теперь любой вводимый символ, не являющийся встроенным, получает контекстную приставку *NewContext‘*. После команды **BeginPackage** следуют заголовки экспортируемых функций, сопровождаемые информацией о назначении каждой функции. Оформляется такая информация в следующем виде:

MyFirstFunction:: usage =
"MyFirstFunction does something..."

Здесь **MyFirstFunction** — заголовок первой экспортируемой функции, а строка содержит информацию, которую можно после подгрузки программы получить стандартным способом, напечатав **?MyFirstFunction** (естественно, слова *does something* следует заменить на более содержательные).

После заголовков экспортируемых функций следует команда

Begin["‘Private‘"]

Следует обратить внимание на то, что слово *Private* окаймлено с двух сторон знаками открытия цитаты. Эта команда, *не меняя контекстной дорожки*, устанавливает новый текущий контекст равным *NewContext‘Private‘*. После выполнения команды все ранее не встречавшиеся символы получат контекстную приставку *NewContext‘Private‘*, означающую, что

они находятся в подконтексте контекста *NewContext*‘. Слово *Private* вполне можно заменить на любое другое, но такова традиция. После команды **Begin["‘Private‘"]** располагаются определения функций, т.е. сама программа. Затем следует команда

End[]

восстанавливающая имевшийся до **Begin["‘Private‘"]** контекст, т.е. контекст *NewContext*‘, и далее команда

EndPackage[]

после которой восстанавливается имевшийся до подгрузки программы текущий контекст (например, *Global*‘), а контекст *NewContext*‘ помещается в начало прежней контекстной дорожки.

Полезно сделать следующие комментарии о работе этого механизма. Поскольку после команды **BeginPackage** на контекстной дорожке помимо контекста программы остается контекст *System*‘, то в случае, когда заголовок какой-либо экспортируемой функции совпадает с заголовком встроенной, экспортируемая функция получает контекстную приставку System‘. Следовательно, вводить функции с заголовками встроенных функций имеет смысл только в том случае, когда новые функции усиливают встроенные, т.е. более эффективно вычисляют какой-то класс выражений или имеют более широкую область применимости.

Явление затенения часто возникает в случаях, когда пользователь напечатал заголовок экспортируемой функции, скажем, **f** до подгрузки программы. Тогда символ *f* получит контекстную приставку текущего контекста, например *Global*‘, и символ *Global*‘*f* затенит символ *NewContext*‘*f* в том смысле, что впечатывание краткого имени *f* будет подразумевать *Global*‘*f*. Но с последним символом не связаны определения экспортируемой функции *NewContext*‘*f*, поэтому выражение

f[arg1,arg2,...] не будет вычисляться ожидаемым образом. Поправить положение можно с помощью функции **Remove**. После вычисления выражения **Remove[f]** символ *Global‘f* будет полностью удален, останется лишь *NewContext‘f*.

9.3. Подгрузка программ

Программы вызываются при помощи команды **Get["name‘"]** или **Needs["name‘"]**, где *name‘* есть контекст программы. Иная форма команды **Get["name‘"]** есть << *name‘*. Разница между командами состоит в том, что **Get** не проверяет, имеется ли контекст программы на контекстной дорожке, а **Needs** проверяет, и при наличии контекста повторной подгрузки не происходит. Поэтому команду **Needs** целесообразно использовать в тех случаях, когда в течение сессии неоднократно подгружались какие-то пакеты. Команда **Needs** читает значение глобальной переменной *$Packages*, которая представляет собой список контекстов вызванных программ. Значение этой глобальной переменной изменяется командой **BeginPackage**. Рассмотренные команды сначала ищут файлы с расширением *m*, в которых хранятся программы. Имеются определенные правила соответствия имен файлов с программами и контекстов программ, зависящие от компьютерной платформы. Для PC-компьютеров название файла без расширения должно содержать не более восьми букв, а контекст программы не должен удовлетворять этому требованию. Поэтому название файла совпадает с первыми восемью буквами контекста с тем исключением, что в контекст могут входить заглавные буквы, которые перейдут в строчные в имени файла. Если контекст программы состоит из контекста и подконтекста, то первые восемь букв контекста составят название директории. Так, программа с контекстом *Algebra‘CountRoots‘* хранится в файле *countroo.m* в директории *Algebra*, поэтому эта программа будет читаться из файла *Algebra\countroo.m*.

Рассмотрим в качестве примера, что происходит с контекстами и контекстной дорожкой при подгрузке программы с контекстом *Algebra‘CountRoots‘* — первой программы в первой поддиректории *Algebra* директории *Packages*. Допустим, что контекстная дорожка в настоящий момент имеет вид {*Global‘*, *System‘*}.

<< Algebra‘CountRoots‘

После выполнения команды

BeginPackage["Algebra‘CountRoots‘"]

текущим контекстом становится строка *Algebra‘CountRoots‘*, а контекстная дорожка принимает вид {*Algebra‘CountRoots‘*, *System‘*}. После рассматриваемой команды в программе идет название вместе с информацией о единственной экспортируемой функции: *CountRoots*:: *usage* = "...", следовательно, символ *CountRoots* получит контекстную приставку *Algebra‘CountRoots‘*, так как нет встроенной функции с тем же заголовком. Таким образом полное имя заголовка будет *Algebra‘CountRoots‘CountRoots*.

Далее выполняется команда

Begin["Algebra‘CountRoots‘Private‘"]

(ее аргументом могла бы быть строка "‘*Private*‘"), устанавливающая текущий контекст *Algebra‘CountRoots‘Private‘*, но оставляющая контекстную дорожку {*Algebra‘CountRoots‘*, *System‘*} неизменной. После этой команды следует *CountRoots*:: *npoly* = "...", являющаяся сообщением об ошибке в случае, если пользователь вместо полинома ввел в качестве аргумента функции **CountRoots** другой объект. Поскольку символ *CountRoots* уже получил контекстную приставку *Algebra‘CountRoots‘* и поскольку контекст *Algebra‘CountRoots‘*

находится на контекстной дорожке, символ *CountRoots* остается с присвоенной ранее контекстной приставкой. Сказанное остается верным относительно этого символа, участвующего в определении

CountRoots[x___] := With[{result = CountRoots0[x]}, result /; result =! = $Failed]

В то же время символы *result* и *CountRoots*0 получают контекстную приставку *Algebra`CountRoots`Private`*, а символы *With* и *$Failed* остаются с контекстной приставкой *System`*. Аналогично символы *OmitZeroes* и *Select*, находящиеся в следующем определении, получают приставки *Algebra`CountRoots`Private`* и *System`* соответственно. Последние в программе четыре заголовка функций *countroots* получают контекстную приставку *Algebra`CountRoots`Private`*. Далее следует команда **End**[], после выполнения которой контекстная дорожка не изменяется, а текущим контекстом становится *Algebra`CountRoots`*. За рассматриваемой командой происходит протектирование функции **CountRoots**, и, наконец, команда **EndPackage**[ ] делает текущим контекстом *Global`*, а контекстную дорожку равной {*Algebra`CountRoots`*, *Global`*, *System`*}. Нелишне добавить, что при чтении программы определения выполняются, поэтому после подгрузки программы соответствующие глобальные правила будут исполняться ядром „Математики“.

Команда **BeginPackage** имеет опциональные аргументы. Их нужно указывать, если в программе используются не только встроенные функции, но также и функции стандартных пакетов или других программ пользователя. Контексты этих программ и указываются в качестве второго и т.д. аргументов функции **BeginPackage**, которая устанавливает их вместе со своим контекстом на контекстной дорожке. Программы, необходимые для работы подгружаемой программы, называются ***импортируемыми***. Эти программы вызываются неяв-

ным использованием функции **Needs**, что исключает повторное подгружение программ. Явное указание импортируемых программ помимо „производственной“ необходимости является признаком хорошего стиля программирования, так как делает все взаимозависимости программ явными и легко прослеживаемыми. Тем не менее указанный способ вызова вспомогательных импортируемых пакетов не всегда удобен. Он имеет то последствие, что контексты вспомогательных программ будут находиться на контекстной дорожке, следовательно, все функции импортируемых пакетов, а не только необходимые для работы программы будут доступны пользователю. Эти функции могут затенять функции, определенные пользователем до подгрузки программы. Избежать этого можно, если импортируемые пакеты вызывать явным указанием их контекстов в команде **Needs**, поставленной сразу после команды **BeginPackage**. В этом случае контексты импортируемых программ не останутся на контекстной дорожке после подгрузки программы, а используемые функции этих программ станут доступны только по их полным символам.

Упражнения

1. Пусть в начале сессии было дано определение f[x_] := 2^x. Затем выполнена команда Begin["a`"] и дано определение g[x_] := 3^x. *Каков* будет *результат* вычисления выражения {f[2], g[2]}? *Каков будет результат* вычисления того же выражения, если вернуться к контексту *Global*`? Будет ли получен результат {4,9}, если после этого вычисления поместить контекст *a*` на контекстную дорожку? *Что* нужно *сделать*, чтобы определение для *g* работало?

2. В начале сессии символу *z* присвоено значение 5, а затем выполнена команда Begin["a`"]; z = 7. *Каковы* будут *результаты* вычисления выражений *z* и a`z?

3. Предположим, что некоему пользователю потребовалось определить функцию strange`Sin, принимающую значение 2 для всех значений аргумента, и он сделал это, написав пакет:

```
(* Странный синус *)
BeginPackage["strange`"]
Sin:: usage = "Sin[x] is unusual function"
Unprotect [Sin]
Begin [" `Private`"]
Sin [x_] := 2
End[]
Protect[Sin]
EndPackage[]
```

Какие результаты даст вычисление выражений **Sin[x]** и **strange`Sin[x]** после подгрузки контекста strange`? *Объясните*, почему получились именно эти ответы.

4. В упражнениях к главам 6 и 7 были определены комбинаторные функции **subsets**, **ksubsets**, **cnt**, **changeto**. *Напишите* пакет с контекстом combin`, в котором определяются рассматриваемые функции. Тем самым эти функции будут всегда доступны вам после подгрузки контекста combin`.

Глава 10
ВВОД И ВЫВОД ДАННЫХ

В этой главе будут рассмотрены вопросы, характеризующие „Математику" как компьютерную среду, в которой возможно не только проводить изолированные символьные, графические и численные расчеты, но и осуществлять структурированный обмен данными с другими составляющими программного, математического и информационного обеспечения исследований. К сожалению, за рамками данной книги останутся все сетевые проблемы, включающие взаимодействие и обмен данными через Интернет. Обсуждение этих вопросов включает изучение языка MathLink, на котором происходит взаимодействие между *интерфейсом* и *вычислительным ядром* „Математики" в версиях под Windows, следующих за 2.2.1. MathLink, в частности, обеспечивает „раздельное" функционирование ядра и интерфейса, что и является предпосылкой сетевого взаимодействия пользователя и вычислительного сервера. Однако в настоящее время соответствующий материал представляется автору недостаточно систематизированным и освещенным в специальной литературе, чтобы его приводить в руководстве для начинающих пользователей.

10.1. Ввод и запись данных в файлы

Электронные Записные книжки представляют собой достаточно удобный инструмент сохранения результатов вычислений между сессиями „Математики". Под результатами здесь понимаются также данные пользователем определения и правила преобразований. Их можно переносить из одной Записной

книжки в другую с помощью копирования и средства Clipboard. Кроме того, оформив надлежащим образом программу с определениями, их можно без нежелательных последствий подгружать по мере возникновения необходимости с помощью команд **Get** и **Needs**, как это обсуждалось в предыдущей главе. Если выходные ячейки содержат результаты длительных вычислений, то, сделав их с помощью **Cell Formatted** и **Cell Active** исполняемыми, можно избавиться от необходимости повторять расчеты, сделанные в предыдущие сессии.

Если необходимо ввести данные из файлов, созданных вне „Математики", то полезно предварительно составить хотя бы общее впечатление о том, как представлены данные в этих файлах. Это можно сделать с помощью команды

!!filename

где *filename* — имя файла. После выполнения этой команды на экране будет представлено содержание файла. Допустим, что цифровые данные записаны в файле *file*1*.val* в следующем виде:

!!file1.val
5 13 7
2 8 3
27 −1 10

Если попытаться подгрузить этот файл с помощью команды **Get**, то мы получим следующий результат:

<< **file1.val**
17

Дело в том, что команда **Get**, предназначенная для ввода выражений, трактует каждую строчку рассматриваемого файла как целостное выражение „Математики", записанное во входном формате, которое после его прочтения вычисляется. При

этом на экран выводится результат вычисления последнего выражения, воспринятого как **Plus[27, Times[−1, 10]]**.

Для чтения данных из файлов следует пользоваться функцией **ReadList**, первым аргументом которой является строка с именем файла, а остальные аргументы опциональные. Если воспользоваться этой функцией без явного указания опций, то придем к результату

ReadList["file1.val"]
{455, 48, 17}

Таким образом, и в этом случае каждая строчка файла воспринята как выражение „Математики", но результаты вычисления каждого выражения помещены в список. Опциональные аргументы функции **ReadList** предназначены для описания типов считываемых данных.

ReadList["file1.val", Number]
{5, 13, 7, 2, 8, 3, 27, −1, 10}

Если данные файла рассматриваются как строки 3×3-матрицы, то нужно воспользоваться опцией *RecordLists → True*, которая по умолчанию установлена на *False*.

ReadList["file1.val", Number, RecordLists → True]
{{5, 13, 7}, {2, 8, 3}, {27, −1, 10}}

Можно добиться того же результата с помощью команды

ReadList["file1.val", {Number, Number, Number}]
{{5, 13, 7}, {2, 8, 3}, {27, −1, 10}}

Последний способ позволяет переструктурировать данные. Например, для того чтобы сгруппировать данные в пары, достаточно выполнить команду:

ReadList["file1.val", {Number, Number}]
{{5,13}, {7,2}, {8,3}, {27,−1}, {10, *EndOfFile*}}

Однако рассматриваемый способ неприменим, если длина строчек в файле неодинакова.

!!file2.val
1.7 3.14
67 2.3*E* − 01 6.2225

ReadList["file2.val", Number, RecordLists → True]
{{1.7,3.14}, {67,0.23,6.2225}}

При этом способе чтения сохранена структура данных в файле. Кроме того, видно, что числовые данные в файле могут быть представлены в так называемой „научной" форме, т.е. с указанием мантиссы и порядка чисел.

Кроме чисел в файле с данными могут находиться и другие объекты, например символы.

!!file3.val
*point*1 0 0 0
*point*2 0 0 1
*point*3 1 1 0

Данные этого файла могут быть считаны с помощью указания следующих опций в команде **ReadList**.

data = ReadList["file3.val",
{Word, Number, Number, Number}]
{{*point*1,0,0,0}, {*point*2,0,0,1}, {*point*3,1,1,0}}

Спецификация типа данных **Word** предполагает, что при считывании символьные данные отображаются в строки.

Head[data[[1,1]]]
String

Таким образом, элементами матрицы, полученной при чтении файла *file3.val*, являются как строки, так и числа. Если требуется строки "*point*1" и т.п. трактовать как символы, то это можно сделать при помощи функции **ToExpression**, которая как раз и преобразует свой аргумент-строку в выражение „Математики“.

data1 = MapAt[ToExpression[#]&, data,
Table[{i, 1}, {i, 3}]];
Head[data1[[1, 1]]]
Symbol

К сожалению, у функции **ReadList** не существует опций, которые позволили бы напрямую считывать числовые данные, если в исходном файле они отделены друг от друга не пробелами, а запятыми. Однако подгрузка таким образом структурированных данных возможна, причем несколькими способами.

!!file4.val
1,2,3
4,5,6

В данном случае можно поступить следующим образом:

data2 = ReadList["file4.val", {Number, Character}]
{{1,,}, {2,,}, {3,}, {4,,}, {5,,}, {6, *EndOfFile*}}

Данные считались парами, на первом месте у которых нужные числа, сопровождаемые запятыми, знаком "\n" перехода на новую строку и символом окончания файла. Извлечение чисел с сохранением матричной структуры теперь не представляет труда.

data3 = Partition[First/@data2, 3]
{{1,2,3}, {4,5,6}}

Самый общий, хотя и весьма трудоемкий способ извлечения нужных данных из файла, — это воспользоваться функцией **Read**, которая подгружает из файла последовательно выражение за выражением. Если при этом что-то нужно пропустить, то следует прибегнуть к функции **Skip**.

!!file5.val
first 1 *second* 100
last 300

Чтобы извлечь из файла только числа, открываем объект „Математики", который называется „входной поток".

stream = OpenRead["file5.val"]
InputStream[*file*5.*val*, 31]

Далее выполняем последовательность команд:

Skip[stream, Word]
n1 = Read[stream, Number];
Skip[stream, Word]
n2 = Read[stream, Number];
Skip[stream, Word]
n3 = Read[stream, Number];
Close[stream]

Извлеченные таким образом числа можно затем поместить в список:

l = {n1, n2, n3}
{1, 100, 300}

Запись данных в файлы в „Математике" не столь гибка, как чтение из файлов. Для записи выражений в файлы служат команды **Put**, **PutAppend** и **Save**. Команда **Put[expr,"file"]**,

или во входном формате **expr >> file**, записывает выражение *expr* в файл *file*.

```
a = x^2 + y^3;
a >> file6.val
!!file6.val
x^2 + y^3
```

Если файл *file6.val* содержал какую-либо другую информацию, то она при записи выражения **a** затирается. Чтобы этого не произошло, следует воспользоваться функцией **PutAppend**.

```
b = Sin[x + y];
b >>> file6.val
!!file6.val
x^2 + y^3
Sin[x + y]
```

Команда **Save** позволяет сохранить в файле правила преобразований, ассоциированные с символом или группой символов.

```
f[1] = 1;  f[x_] := x/; x > 0;  f[x_] := -x/; x < 0
Save["file7.val", f]
!!file7.val
f[1] = 1
f[x_] := x/; x > 0
f[x_] := -x/; x < 0
```

Функция **WriteString** используется для записи строк в файлы. Использование этой функции предполагает предварительное открытие „выходного потока“.

```
stream = OpenWrite["file8.val"]
OutputStream[file8.val,]
```

Заметим, что предварительное существование файла *file8.val* не предполагается, он создается функцией **OpenWrite**. Рассмотрим пример, где требуется сохранить матрицу

m = {{1,2}, {a,b}};

Предварительно конвертируем ее в строку следующего вида:

mstring = ToString[TableForm[m,
TableSpacing → {0,1}]]
1 2
a b

и записываем *mstring*:

WriteString[stream, mstring]
Close[stream];
!!file8.val
1 2
a b

10.2. Обмен данными с другими программами

В „Математике" предусмотрены средства, позволяющие отформатировать выражения так, чтобы они стали непосредственно доступными для таких программ, как TeX, C и Fortran. Соответствующими функциями являются **TeXForm**, **CForm** и **FortranForm**. Их следует применить к тому выражению, которое вы хотите записать в файл для обработки нужными программами.

l = {c, d}^2/3

$$\left\{\frac{c^2}{3}, \frac{d^2}{3}\right\}$$

ltex = TeXForm[l]
\{ {{c^2} \over 3}, {{d^2} \over 3} \}

Выражение **ltex** можно записать или добавить в соответствующий файл

ltex >>> file9.tex

и после обработки транслятором TeX'а получить в тексте формулу в том виде, как она представлена на экране (в данном случае). Посмотрим теперь, как выглядит вычисленное выражение l в C- и Fortran-форматах.

lc = CForm[l]
List(*Power*(*c*,2)/3, *Power*(*d*,2)/3)

lfortran = FortranForm[l]
List(*c* **2/3, *d* **2/3)

Выражения **lc** и **lfortran** можно поместить в соответствующие C- или Fortran-файлы.

Еще одна интересная возможность взаимодействия „Математики" и рассматриваемых программ предоставляется функцией **Splice**. Можно в файлы, содержащие программы на языках C или Fortran, а также в TeX-файлы вставлять выражения „Математики", ограничивая их слева и справа с помощью знаков < *, * >, и использовать „Математику" для того, чтобы вычислить вставленное выражение и поместить результат на место исходного выражения. Для этого нужно программу, содержащую выражения „Математики", поместить в файл с расширением mc, mf или mtex и обработать этот файл с помощью функции **Splice**, т.е. вычислить выражение **Splice["file.mx"]**. Обработанная программа будет помещена в файл с именем *file.x*, где *x* стоит вместо *c*, *f* или *tex*. Вот соответствующий пример Fortran-программы со вставленным выражением „Математики".

!!fort.mf

```
example
real   x, y
x = 1.
y =< *Integrate[x^6/(x^4 + 1), x]* >
write(*,*) y
stop end
```

Splice["fort.mf"];

!!fort.f

```
example
real   x, y
x = 1.
y = x**3/3 - ArcTan((-Sqrt(2)+2*x)/Sqrt(2))/(2*Sqrt(2)) -
   ArcTan((Sqrt(2)+2*x)/Sqrt(2))/(2*Sqrt(2)) -
   Log(1-Sqrt(2)*x+x**2)/(4*Sqrt(2)) +
   Log(1+Sqrt(2)*x+x**2)/(4*Sqrt(2))
write(*,*)y
stop
end
```

10.3. Форматирование выходных ячеек

В первой главе мы рассматривали три формата выражений „Математики“: входной, в котором выражение записывается в одну строчку с использованием, если удобно, префиксной, инфиксной и постфиксной форм функций „Математики“, внутренний и выходной, значительно более, чем входной приближающийся к обычной форме математических выражений. Здесь мы обсудим различные приемы, позволяющие устанавливать специальные выходные форматы результатов вычислений, т.е. форматы выходных ячеек. Отметим, что вычисляющее ядро „Математики“ имеет дело только со внутренней, или полной, формой выражений, такой, как она может быть пред-

ставлена функцией **FullForm**. Следовательно, форматирование выходных данных никак не влияет на ход и результаты вычислений: оно применяется к уже готовому результату.

Существуют шесть стандартных форматов: InputForm, OutputForm, CForm, FortranForm, TeXForm и TextForm — для представления результатов на экране дисплея. Таким образом, выходных форматов пять, и в первой главе мы объединили в одно понятие пять различных форматов, три из которых уже были рассмотрены в предыдущем параграфе. Заметим, что результаты вычислений могут быть представлены в выходной ячейке и во входном формате. Для отдельной ячейки это достигается с помощью Cell Formatted, однако можно установить этот формат и для всей Записной книжки. Строка “stdout“ есть имя специального выходного “потока“, зарезервированного для дисплея. Этот поток имеет опции, значения которых можно узнать обычным образом.

Options["stdout"]
{FormatType → OutputForm, PageWidth → 61,
PageHeight → 22, TotalWidth → Infinity,
TotalHeight → Infinity,
StringConversion → $StringConversion}

Переопределив опцию FormatType на InputForm, получим в выходных ячейках результаты, представленные во входном формате.

SetOptions["stdout", FormatType → InputForm];
Expand[(x + 2y)^2]
*x^2 + 4 * x * y + 4 * y^2*

К глобальному изменению выходного формата приходится прибегать довольно редко. Гораздо чаще возникает потребность в изменении стандартного формата для отдельных выражений или классов выражений. Например, часть или все переменные

какой-то функции удобно писать в виде верхних или нижних индексов (тензоры); переменные, по которым берутся частные производные, также удобно писать в виде индексов и т.д. Подобно **OwnValues**, **DownValues** и **UpValues**, с символами можно ассоциировать **FormatValues**. Эта ассоциация осуществляется функцией **Format**.

Предположим, что f есть заголовок функции с неопределенным числом аргументов, и, чтобы не загромождать экран, мы хотели бы не печатать аргументов функции f, если их число превосходит два. Тогда мы можем следующим образом отформатировать представление выражения с заголовком f на экране:

```
Format[f[x__]] := f/; Length[{x}] > 2
f[x,y,z,w]
f
% // FullForm
f[x,y,z,w]
f[x,y]
f[x,y]
FormatValues[f]
Literal[f[x__]] :> f/; Length[{x}] > 2
```

Как показывает результат, полученный при вычислении выражения **% // FullForm**, форматирование влияет только на графическое представление выражения на экране, но не влияет на значение вычисленного выражения. Если возникает необходимость отменить введенный формат, то следует прибегнуть к команде **FormatValues[f] = .**

При форматировании отдельных выражений или классов выражений очень удобны функции **StringForm** и **SequenceForm**. Будучи значением функции **Format**, выражение

StringForm["string1‘‘string2‘‘...,expr1,expr2,...]

где *string*1, *string*2 и т.д. последовательности символов, которые могут включать пробелы, знаки \n перехода на новую строчку и другие знаки форматирования, а между знаками ‘ вставляются вычисленные значения выражений *expr*1, *expr*2 и т.д.

Format[f[x__]] := StringForm["‘‘[‘‘arguments]", f,
Length[{x}]]/; Length[{x}] > 3
f[x,y,z,u,v,w]
f[6 *arguments*]

Если выражение **SequenceForm[expr1,expr2,...]** является значением функции **Format**, то в выходной ячейке соответствующее выражение, являющееся аргументом функции **Format**, будет представлено как конкатенация вычисленных выражений **expr1, expr2** и т.д.

SequenceForm[a, b + 1, c]
$a1 + bc$

При проведении математических расчетов применение функции **SequenceForm** эффективно в сочетании с функциями, обеспечивающими представление части аргументов как верхних или нижних индексов.

{Subscripted[f[x1,x2,x3], Subscripted[f[x1,x2,x3],2],
Subscripted[[f[x1,x2,x3],{1,2},{3}]]}
$\{f_{x1,x2,x3},\ f_{x1,x2}[x3],\ f_{x1,x2}{}^{x3}\}$

Как видно из примеров, выражение

Subscripted[f[x1,x2,...],{n1,n2},{m1,m2}]}

при вычислении записывает аргументы функции **f** с номерами от $n1$ до $n2$ как нижние, а с номерами от $m1$ до $m2$ — как

верхние индексы. Функции **Subscript** и **Superscript** отсылают свои аргументы соответственно в нижние и верхние индексы.

{Subscript[a, b], Superscript[a, b]}

$\{_{ab},\ ^{ab}\}$

При форматировании тензоров последние функции особенно удобны, поскольку они убирают запятые между последовательными аргументами.

Format[f[x1, x2, x3]] := SequenceForm[f,
Subscript[x1, x2], Superscript[x3]]
f[x1, x2, x3]

$f_{x1x2}{}^{x3}$

В качестве более сложного примера рассмотрим форматирование частных производных функций такое, что если производная от функций вычисляется только в символьном виде, то переменные, по которым вычисляются производные, записываются как нижние индексы. Например, выражение

zz = D[f[t, x, y], {t, 2}, x, y, x]

$f^{(2,2,1)}[t, x, y]$

должно быть представлено на экране как

$f_{ttxxy}[t, x, y]$

Уточним, что так должны быть представлены любые производные от любых функций от произвольных аргументов.

Для решения задачи напишем функцию **preformat**, которая, будучи примененной к выражениям вида **zz**, создает нужное представление. Воспользуемся тем, что полная форма **zz** имеет вид

zz // FullForm

Derivative[2, 2, 1][*f*][*t*, *x*, *y*]

и будем извлекать из этой полной формы блоки искомого представления. Сначала извлечем аргументы функции:

```
za = List@@zz
{t, x, y}
```

Затем извлекаем индекс (2,2,1):

```
zb = List@@Head@Head@zz
{2, 2, 1}
```

С помощью выражений **za** и **zb** сконструируем вспомогательный список

```
zc = Transpose[za, zb]
{{t, 2}, {x, 2}, {y, 1}}
```

необходимый, чтобы получить последовательность аргументов, по которым вычисляется производная, в виде *ttxxy*.

```
zd = (cc = First[#]; cc& /@Range[Last[#])& /@zc //
Flatten
{t, t, x, x, y}
```

К каждому элементу списка **zd** мы впоследствии применим функцию **Subscript**, чтобы они сформировали нижний индекс. Заголовок функции, производная которой вычисляется, есть значение выражения

```
ze = First@Head@zz
f
```

С помощью **SequenceForm** получаем заголовок отформатированной производной:

```
zf = SequenceForm[ze, Sequence@@Subscript /@zd]
```
f_{ttxxy}

который применяем к **za**:

```
zf @@ za
```
$f_{ttxxy}[t,x,y]$

Оформляем промежуточные вычисления в виде функции:

```
preformat[z_/; Nest[Head, z, 3] ===
Derivative] := Module[{aa, ab, cc},
aa = Transpose[{List @@ z, List @@ Head @ Head @ z}];
ab = Subscript /@ (cc = First[#];
cc& /@ Range[Last[]#])& /@ zc // Flatten; ac =
SequenceForm[First @ Head @ z, Sequence @@ ab];
ac @@ List @@ z]
```

Наконец, форматируем всевозможные производные:

```
Format[z: (Derivative[__][_][__])] := preformat[z]
```

Проверим, как работает форматирование:

```
D[h[x, y], {x, 3}, y]
```
$h_{xxxy}[x,y]$

Упражнения

1. Пусть в результате занявшего много времени расчета в выходную ячейку помещен список численных данных. Как воспользоваться ими, не повторяя расчета? *Смоделируйте* эту ситуацию следующим образом. Откройте Записную книжку и вычислите выражение **Sin[Range[100]] // N**. Сохраните Записную книжку, начните новую сессию „Математики" и вновь откройте эту Записную книжку. Присвойте символу *l* значение выражения в выходной ячейке, предварительно изменив ее свойства, и проверьте справедливость равенства **l^2 == 1 – Cos[Range[100]]^2 // N**.

2. Транспортная компания получила следующий документ о категориях грузов и районах их доставки:

Категория груза	5	5	3	2	2	3
Район доставки	U1	R2	R1	R2	Ltv	U2
Категория груза	5	4	3	1	2	5
Район доставки	U2	R3	R3	U1	U2	R3

Пусть этот документ находится в файле transport.val. *Введите* данные из него в „Математику" и нарисуйте гистограммы числа грузов каждой категории и числа грузов в каждый район доставки.

3. *Определите* выходной формат $Li_n[z]$ полилогарифмической функции **PolyLog[n,z]**. В выходной ячейке она должна быть представлена в этом виде в тех случаях, когда не выполняется ее вычисление в явном виде.

4. Пусть тензор Риччи $R_{ij}{}^{kl}$ определен функцией **Ricci[i,j,k,l]**. *Отформатируйте* эту функцию так, как она традиционно пишется в книгах, в частности в этой.

5. *Определите* логическую функцию **Implication[x,y]**, которая для булевых x, y принимает значение True, за исключением случая $x = True$, $y = False$. *Создайте* выходной формат, такой, чтобы в случаях, когда аргументы не вычисляются на True или False, в выходной ячейке содержалось бы выражение $x \to y$ с вычисленными x и y. *Выясните*, как будет представлен результат вычисления выражения

Implication[LogicalExpand[a||!a], True]

ОТВЕТЫ И РЕШЕНИЯ К УПРАЖНЕНИЯМ

Глава 2

1. Обозначим через a многочлен в левой части уравнения и делаем последовательность преобразований:

   $b = a/x^2$ // Apart,

   $c = b/.(1/x \to y - x)$ // Expand,

   $d = c/.(1/x^2 \to y^2 - x^2 - 2)$,

   $s = \text{Solve}[d == 0, y]$,

   $sx1 = \text{Solve}[x + 1/x == y/.s[[1,1]], x]$,

   $sx2 = \text{Solve}[x + 1/x == y/.s[[2,1]], x]$

2. Примените Factor.

3. Найдите CoefficientList многочлена $x^2 + y^2 + z^2 - (ax + by + cz)(Ax + By + Cz)$ // Expand и решите с помощью Solve соответствующую систему уравнений.

4. Пусть $int = Integrate[1/(2 + Cos[x]), x]$. Нарисуйте график int на отрезке от -2π до 2π с помощью функции Plot.

5. Пусть $e = xy\hat{}3 - (x+1)\hat{}2y + 3 == 0$, тогда $ex = \text{Dt}[e, x]$ является линейным уравнением относительно $\text{Dt}[y, x]$, а $exx = \text{Dt}[ex, x]$ линейным уравнением относительно $\text{Dt}[y, \{x, 2\}]$. Разрешая ex относительно $\text{Dt}[y, x]$ и подставляя $x = 0$, $y = 3$, находим, что $y'(x) = 21$. Из второго уравнения получаем $y''(x) = 1044$.

6. Пусть $e = z\hat{}5 - xz + y\hat{}2\ z\hat{}2 - 1 == 0$, тогда $ex = \text{Dt}[e, x, Constants \to \{y\}]$ является линейным уравнением относительно $\text{Dt}[z, x, Constants \to \{y\}]$, откуда находим, что $z_x(0,0) = \text{Solve}[ex,\ \text{Dt}[z,\ x,\ Constants \to \{y\}]][[1,1,2]]/.\{x \to 0, y \to 0, z \to 1\} = 1/5$. Аналогично находим, что $z_y(0,0) = 0$.

7. Рисуем заданные кривые, вычисляя выражение

   ImplicitPlot[{4y^2 == (x - 1)^5, y^2 == x}, {x, 0, 3}].

Находим, что абсцисса точек пересечения кривых приближенно равна 2.5. Уточняем ее значение x_0 с помощью функции FindRoot: $x_0 = 2.59701$. Обозначая $Sqrt[1 + D[1/2(x-1)\hat{}(5/2), x]\hat{}2]l$, вычисляем $NIntegrate[l, \{x, 1, 2.59701\}]$ и получаем длину половины дуги, равную 2.43193.

Глава 3

3. ImplicitPlot
5. Одно из возможных решений:

```
hyp = Table[ParametricPlot[{Cosh[u]Cos[t], Sinh[u]Sin[t]}, {u, -1, 1},
PlotRange -> {{-1.2, 1.2}, {-1, 1}}, AspectRatio ->
Automatic], {t, 0.1, Pi + 0.1, 0.2Pi}];
ellip = Table[ParametricPlot[{Cosh[u]Cos[t], Sinh[u]Sin[t]}, {t, 0, 2Pi},
AspectRatio -> Automatic, PlotRange ->
{{-1.2, 1.2}, {-1, 1}}], {u, 0.1, 0.5, 0.1}];
Show[hyp, ellip];
```

6. Пример: Show[Graphics3D[MoebiusStrip[ ]], Boxed → False];
7. Пример: Show[Polyhedron[GreatIcosahedron], Boxed → False]
8. Show[Graphics[Line[{{2,0}, {0,0}, {0,2}, {2,2}, {2,0}, {0,2}, {1,3}, {2,2}, {0,0}}]], AspectRatio → Automatic]

Глава 4

1. l3 = Last[Transpose[Partition[l, 3]]],
   Delete[l, Table[{3i}, {i, Length[l]/3}]]
2. Delete[l, Position[l, 0]]
3. Plus @@ Select[l, Negative]
4. Plus @@ (l^2)/Length[l]
5. Min[Abs[l]]
6. Position[l, First[Select[l, Negative]]]
7. Or @@ PrimeQ /@ l
8. Transpose[Insert[Transpose[m], l, k]]
9. Table[j^2, {i, 0, 4}, {j, 0, i}]

10. Сначала порождаем список

```
t = Table[{2 Random[Integer] - 1, 2 Random[Integer] - 1}, {i, 20}],
```

затем из *t* получаем

```
m = FoldList[Plus, 0, t] // Rest
```

11. `Det[Outer[D, {x + y^3, x^2 - y^2 - y}, {x, y}] /. {x -> 0, y -> 0}] = -1`

12. `Outer[D, {x y + z}, {x, y, z}] // First = {y, x, 1}`

13. `Inner[D, {f[x, y, z], g[x, y, z], h[x, y, z]}, {x, y, z}]`

14. `Inner[Rule, l, m, List]` или `Thread[Rule[l, m]]`

15. `Thread[Equal[m.l, r]]`

Глава 5

1. `perfectQ[x_] := Apply[Plus, Divisors[x]] === 2 x`

2.
```
CardDeck = Flatten[Outer[List, {s, c, h, d}, Join[{A, K, D, J},
Range[7, 10]]], 1]
```

3. `perfectshuffle[x_, n_] := Nest[Flatten[Transpose[Partition[#, 16]], 1]&, x, n]`

4.
```
deleterand[x_, n_] := Nest[Delete[#, Random[Integer, {1, Length[#]}]]&, x, n]
deal[z_] := Function[{x}, Apply[Complement[#1, #2]&, x]] /@
Partition[NestList[deleterand[#1, 8]&, z, 4], 2, 1]
```

5. `tobase[n_, b_] := Fold[(10#1 + #2)&, 0, IntegerDigits[n, b]]`

6. `Union[Select[l, Count[l, #] > 1&]]`

7. `Flatten[Position[l, #]& /@ Select[l, PrimeQ], 2]`

8. `{#, Count[l, #]}& /@ l // Union`

9.
```
Outer[Dot, Array[w, 4], Array[w, 4]] /. Thread[Array[w, 4] -> l] =
{{4, 10, 3, 11}, {10, 30, 7, 35}, {3, 7, 5, 1}, {11, 35, 1, 59}}
```

10.
```
v = Last[l] - LinearSolve[Transpose[Delete[Transpose[Delete[gs, 4]], 4]],
Take[Last[gs], 3]] . Delete[l, 4]
```

Глава 6

1. `l1 = l //. {z_, u___, {x__}, v___} :> {z, u, {0, x}, v} /; Length[{x}] < 4`

2. `wronsk[l_List, t_Symbol] := Det[NestList[D[#, t]&, l, Length[l] - 1]]`

3. Однострочник: ord[x_] := Plus@@Head@Head[x].
   Локальные правила: ord[x_] := x/. Derivative[z__][_][__] :> Plus[z]

4. newton[f_, x0_, eps_] := ({z1, z2} = {N[x0] - f[N[x0]]/f'[N[x0]], N[x0]}; {z1, z2} //. {a_, b_} :> {a - f[a]/f'[a], a} /; Abs[b - a] > eps)

5. crossproduct[u_List, v_List] /; Length[u] == 3 && Length[v] == 3 := First[Reverse/@Minors[{u, v}, 2]]

6. dirderiv[u_List, f_] /; Length[u] == 3 := u . Through[(Function[{r}, D[#1, r]&]/@{x, y, z})[f[x, y, z]]]

7. decode[l_List /; Depth[l] == 3] := Flatten[l //. {x___, {a_, k_Integer}, y___} :> {x, Table[a, {k}], y}]

8. subsets[l_List] /; Length[l] > 1 :=
   Join[subsets[Rest[l]], Join[{First[l]}, #1]& /@subsets[Rest[l]]]
   subsets[{x_}] := {{}, {x}}

9. ksubsets[l_List, (k_Integer)?Positive] /; Length[l] > k && k > 1 :=
   Union[ksubsets[Rest[l], k], (Append[#, First[l]]&)/@ksubsets[Rest[l], k - 1]]
   ksubsets[l_, k_] /; Length[l] == k := {l}
   ksubsets[l_, 1] := {#}& /@l

Глава 7

1. fact[n_] := Module[{s}, If[Head[n] === Integer && Positive[n], s = 1; Do[s = s * i, {i, n}]; s, Print["the argument is not a positive integer"]]]

2. fib[n_] := Module[{r, s}, If[Head[n] === Integer && Positive[n], r = 1; s = 0; Do[{r, s} = {r + s, r}, {n - 1}]; r, Print["the argument is not a positive integer"]]]

3. numberofprimes[n_] := Module[{s = 1}, While[Prime[s] <= n, s = s + 1]; s - 1]

4. cnt[l_List] := Module[{l1 = Union[l]}, Do[For[i = 1; s[j] = 0, i <= Length[l], i++, If[l[[i]] === l1[[j]], s[j] = s[j] + 1]], {j, Length[l1]}]; Transpose[{l1, Array[s, Length[l1]]}]]

5. changeto[n_Integer, x_List] := Module[{r, o}, r =
   (Range[0, #]&)/@(Quotient[n, #]&)/@x;
   o = Flatten[Outer[List, Apply[Sequence, r]], Length[x] - 1];
   Select[o, x . # === n&]]

6. latinsquare[(n_Integer) ? Positive] := Module[{difQ, p, e},
   difQ[x_List, y_List] := Apply[And, Inner[UnsameQ, x, y, List]]; p =
   Permutations[Range[n]]; e = p[[Random[Integer, {1, Length[p]}]]];
   differ[z_List, u_] := Select[z, difQ[#, u]&]; f[{z_List, v_List}] :=
   {differ[z, First[v]], Prepend[v, differ[z, First[v]]
   [[Random[Integer, {1, Length[differ[z, First[v]]]}]]]]};
   Nest[f, {p, {e}}, n – 1][[2]]]

7. orthQ[x_List, y_List] := Module[{m, n},
   m = MapThread[Inner[List, #1, #2, List]&, {x, y}];
   n = Flatten[m, 1]; Length[n] == Length[Union[n]]]

Глава 8

1. Apply[List, Hold[1 + 2 + 3], 2] // ReleaseHold
2. SetAttributes[f, HoldFirst]; f[x_Plus] :=
   Apply[List, Hold[x], 2] // ReleaseHold
3. SetAttributes[Information, Listable]
5. Вычисление $f[3]$ можно смоделировать выражением

   $ReleaseHold[Hold[(x = N[x]; Sqrt[x])] /. x \to 3]$,

   поэтому вычисление выражения $3 = N[3]$ приводит к сообщению об ошибке. Определение можно изменить, например, так:

   $f[x_] := (x1 = N[x]; Sqrt[x1])$.

Глава 9

1. В первом случае {4,9}, во втором — {4, g[2]}. Результат {4,9} не будет получен, так как символу g будет присвоена приставка Global`. Для того чтобы определение для g работало, нужно вычислить выражение {f[2], ag[2]}.
2. Результаты вычислений: 7 и z. Несмотря на то что текущим контекстом при вычислении выражения z = 7 был a`, значение 7 было присвоено символу *Global`z*, так как контекст Global` находился на контекстной дорожке. Символ *a`z* появился только при выполнении последнего вычисления.

3. Результаты: 2 и *Sin*[*x*]. При подгрузке пакета контекст System\` находился на контекстной дорожке, поэтому экспортируемая функция Sin не получила новой контекстной приставки, оставшись со старой приставкой System\`. В силу того что определения пользователя имеют приоритет перед системными определениями, результат вычисления функции **Sin** получился равным двум. С функцией **strange\`Sin** не связано никаких определений, поэтому выражение **strange\`Sin[x]** осталось фактически не вычисленным. Контекстная приставка не отобразилась на дисплее для **strange\`Sin**, так как в „чужом" для обеих функций контексте Global контекст *strange*\` имеет преимущество, находясь на контекстной дорожке левее контекста System\`.

Глава 10

2.
```
r1 = ReadList["transport.val", Word]
r2 = ToExpression /@ Transpose[Drop[Transpose[Partition[r1, 8]], 2]] //
Flatten
r3 = {Count[r2, #], #, } & /@ Union[r2]
data1 = Cases[r3, {_Integer, _Integer}]
data2 = Cases[r3, {_, _Symbol}]
<< Graphics`Graphics`
BarChart[data1]; BarChart[data2];
```

3.
```
Unprotect[PolyLog]
Format[PolyLog[n_, z_]] := Subscripted[Li[n, z], 1]
Protect[PolyLog]
```

4.
```
Format[Ricci[i_, j_, k_, l_]] := SequenceForm[R, Subscript[i, j], Superscript[k, l]]
```

5.
```
Implication[x_, y_] /; MemberQ[{True, False}, x] &&
MemberQ[{True, False}, y] := !x || y
Format[Literal[Implication[x_, y_]]] := StringForm["  “ → “ ", x, y]
```

КРАТКИЙ СПРАВОЧНИК ПО ВСТРОЕННЫМ ФУНКЦИЯМ „МАТЕМАТИКИ“

Алгебраические преобразования

Apart[**expr**] представляет рациональное выражение в виде суммы дробей с минимальными знаменателями; *Apart*[*expr*, *var*] рассматривает все переменные, кроме *var* как константы, и записывает *expr* как полином по *var* совместно с суммой отношений полиномов, в которых степень по *var* каждого полинома в числителе меньше, чем степень знаменателя. Apart[expr, Trig → True] рассматривает тригонометрические функции как рациональные функции от степеней *e*.

Cancel[**expr**] сокращает общие множители в числителе и знаменателе рационального выражения *expr*. При наличии опции Trig → True рассматривает тригонометрические функции как рациональные функции от степеней *e*.

Coefficient[**poly**, **form**] дает коэффициент при *form* в полиноме *poly*, причем *form* может быть и произведением. Coefficient[poly, form, n] дает коэффициент при $form^n$, a Coefficient[poly, form, 0] выбирает члены, которые не пропорциональны *form*.

CoefficientList[**poly**, **var**] дает список коэффициентов при степенях *var* в *poly*, начиная с нулевой степени. CoefficientList[poly, {var_1, var_2, ...}] дает матрицу коэффициентов var_i. Размерности матрицы коэффициентов определяются значениями Exponent[poly, var_i]. Члены, не содержащие положительных степеней какой-либо переменной, включаются в первый элемент списка для этой переменной. CoefficientList всегда имеет своим значением прямоугольную матрицу. Комбинации степеней, не содержащиеся в *poly* дают нули матрицы.

Collect[**poly**, **x**] группирует члены с одной и той же степенью переменной x. Collect[*poly*, {x_1, x_2, ...}] группирует члены с одними и теми же степенями переменных x_1, x_2, ... „Переменные“ x_i сами могут быть произведениями. Collect[*poly*, Trig → True] рассматривает тригонометрические функции как рациональные выражения от степеней *e*.

ComplexExpand[expr] раскладывает *expr* в предположении, что все переменные вещественные. ComplexExpand[*expr*, {x_1, x_2, ...}] раскладывает *expr* в предположении, что все переменные, отвечающие x_i, комплексные. Опция TargetFunction состоит из функций списка {Re, Im, Abs, Arg, Conjugate, Sign}. При наличии этой опции ComplexExpand приводит к результатам с максимально возможным использованием функций, указанных в опции.

Decompose[poly, x] — список более простых полиномов P_1, P_2, ..., композицией которых $P_1(P_2(\ldots(x)\ldots))$ является *poly*. Список полиномов P_i не является единственно возможным. Декомпозиция есть операция, не зависимая от разложения на множители.

Denominator[expr] — знаменатель *expr*. Denominator извлекает члены с явно отрицательными степенями. Numerator — все остальные. Степень считается „явно отрицательной", если она имеет множителем отрицательное число. Стандартное представление рациональных выражений как произведений степеней означает то, что нельзя просто использовать Part для того, чтобы извлекать знаменатели. Denominator можно использовать для получения знаменателей рациональных чисел.

Expand[poly] раскрывает произведения и положительные степени, в то время как Expand[*poly*, *pattern*] избегает раскладывать элементы *poly*, которые не содержат члены, соответствующие шаблону *pattern*. Expand работает только с целыми положительными степенями и применяется только к верхнему уровню выражений. Expand[*expr*, Trig → True] трактует тригонометрические функции как рациональные функции степеней *e* и соответственно с этим их раскрывает.

ExpandAll[expr] раскрывает произведения и целые степени в любых частях выражения *expr*, в то же время ExpandAll[*expr*, *pattern*] избегает раскладывать те части выражения *expr*, которые не содержат членов, соответствующих шаблону *pattern*. ExpandAll[*expr*] применяет фактически Expand и ExpandDenominator к каждой части *expr*. ExpandAll[*expr*, Trig → True] трактует тригонометрические функции как рациональные выражения от степеней *e* и соответственно их раскладывает.

ExpandDenominator[expr] раскрывает произведения и степени в знаменателях выражения *expr*. Эта функция применяется только к отрицательным целым степеням и только к верхнему уровню выражений.

ExpandNumerator[expr] раскрывает произведения и степени в числителях выражения *expr*. Эта функция применяется только к целым положительным степеням и только к верхнему уровню выражений.

Exponent[expr,form] дает максимальную степень,с которой *form* входит в *expr*. Exponent[*expr*, *form*, *h*] применяет *h* к множеству показателей, с которыми *form* входит в *expr*. По умолчанию, *h* есть Max. Выражение *form* может быть произведением. Относительно *expr* предполагается, что это отдельный член или сумма членов. Exponent применяется только к верхнему уровню *expr*.

Factor[poly] раскладывает полином на множители над кольцом целых чисел. Factor[*poly*, Modulus → *p*] раскладывает полиномы по модулю простого числа *p*. Factor[*expr*, Trig → True] рассматривает тригонометрические функции как рациональные функции от степеней числа *e* и раскладывает их.

FactorList[poly] дает список множителей полинома *poly* вместе с показателями степеней, с которыми они входят в разложение *poly* на множители. Первый элемент списка есть общий численный множитель, а если такого отличного от единицы нет, то список начинается с {1,1}.

FactorSquareFree[poly] записывает полином как произведение степеней свободных от квадратов сомножителей.

FactorSquareFreeList[poly] дает список всех свободных от квадратов сомножителей полинома *poly* вместе с их показателями.

FactorTerms[poly] — общий числовой множитель в *poly*, а FactorTerms[*poly*, *x*] выносит общий множитель, не зависящий от *x*. FactorTerms[*poly*, $\{x_1, x_2, \ldots\}$] последовательно выносит множители в *poly*, которые не зависят от каждого x_i.

FactorTermsList[poly, $\{x_1, x_2, \ldots\}$] — список множителей в *poly*. Первый элемент в списке есть общий числовой множитель, второй — множитель, не зависящий ни от одного из x_i. Последующие элементы есть множители, не зависящие от как можно большего числа x_i.

Length[poly] — число слагаемых в *poly*.

Numerator[expr] — числитель *expr*. Numerator выделяет члены, не имеющие явно отрицательных показателей. Denominator выделяет остальные. Степень считается „явно отрицательной", если она имеет отрицательное число множителем. Numerator можно использовать для извлечения числителя рационального числа.

PolynomialGCD[$poly_1$, $poly_2$] — наибольший общий делитель полиномов $poly_1$ и $poly_2$. PolynomialGCD[$poly_1$, $poly_2$, Modulus → *n*] находит наибольший общий делитель по модулю целого *n*.

PolynomialLCM[$poly_1$, $poly_2$] — наименьшее общее кратное полиномов $poly_1$ и $poly_2$. PolynomialLCM[$poly_1$, $poly_2$, Modulus → *n*] находит НОК по модулю целого *n*.

PolynomialMod[poly,m] — *poly*, редуцированное по модулю m. В случае задания вместо m списка $\{m_1, m_2, \ldots\}$ редуцирует по модулю всех m_i.

PolynomialQ[expr,var] имеет значение True, если *expr* есть полином по переменной *var*, и False — в противном случае. PolynomialQ[*expr*, {var_1, var_2,...}] проверяет, является ли *expr* полиномом по var_i. Выражения var_i не обязательно символы. PolynomialQ[*expr*] проверяет, является ли *expr* полиномом относительно каких-либо переменных. Результат False может быть, например, в случае, когда *expr* содержит числа типа Real.

PolynomialQuotient[p,q,x] дает результат деления p на q, рассматриваемых как полиномы по переменной x, с отбрасыванием остатка.

PolynoialRemainder[p,q,x] — остаток от деления p на q, рассматриваемых как полиномы по переменной x.

PowerExpand[expr] раскрывает все степени произведений. PowerExpand преобразует (*ab*)^*c* в *a*^*c* *b*^*c* и (*a*^*b*)^*c* в *a*^(*b* *c*), какого бы вида ни было *c*. Преобразования, сделанные с помощью PowerExpand, корректны в общем случае, только если *c* есть целое, а *a* и *b* — положительные вещественные числа.

Resultant[poly₁,poly₂,var] вычисляет результант полиномов $poly_1$ и $poly_2$ по отношению к переменной *var*. Resultant[$poly_1$,$poly_2$,*var*, Modulus → *p*] вычисляет результат по модулю простого *p*.

Simplify[expr] выполняет последовательность алгебраических преобразований над *expr* и приводит его к простейшей форме. При наличии опции Trig → False тригонометрические тождества не используются.

Together[expr] приводит выражение *expr* к общему знаменателю и сокращает общие множители в числителе и знаменателе. При наличии опции Trig → True тригонометрические функции трактуются как рациональные функции от степеней *e*.

Variables[poly] — список всех независимых переменных в полиноме *poly*.

Графика

ContourPlot[f,{x,xmin,xmax},{y,ymin,ymax}] рисует контурный график f как функции переменных x и y.

DensityPlot[f, {x,xmin,xmax}, {y,ymin,ymax}] рисует плотностный график f как функции переменных x и y.

ListContourPlot[l] рисует контурный график по массиву высот l, который должен представлять собой прямоугольную матрицу. Каждая высота соответствует точке с целочисленными координатами (m, n), где m и n изменяются начиная с 1.

ListDensityPlot[l] рисует плотностный график по массиву высот l, который должен представлять собой прямоугольную матрицу. Каждая высота соответствует точке с целочисленными координатами (m, n), где m и n изменяются начиная с 1.

ListPlot[l] — график дискретных данных. Если l есть список чисел, то каждое из них интерпретируется как значение ординаты в точке с абсциссой, равной номеру числа в списке. Возможно задание l в виде списка пар чисел, и тогда первое число пары трактуется как значение абсциссы, а второе — ординаты точки.

ListPlot3D[l] рисует двухмерную поверхность, представляющую массив высот l. Каждое число в l интерпретируется как высота в точке целочисленной решетки (m, n), где m и n изменяются, начиная с 1.

ParametricPlot[{f,g},{t,tmin,tmax}] рисует кривую на плоскости, заданную параметрически с помощью f и g. Возможно одновременное представление нескольких кривых, если первый аргумент есть список пар функций. Параметр t изменяется в пределах от $tmin$ до $tmax$.

ParametricPlot3D[{f,g,h},{t,tmin,tmax}] — пространственная кривая, заданная параметрически функциями f, g и h. Возможно одновременное представление нескольких кривых в случае, когда первый аргумент есть список троек функций. Параметр t изменяется от $tmin$ до $tmax$. ParametricPlot3D[{f,g,h},{x,$xmin$,$xmax$},{y,$ymin$,$ymax$}] рисует двумерную поверхность, параметризованную переменными x и y, изменяющимися от $xmin$ до $xmax$ и от $ymin$ до $ymax$ соответственно.

Plot[f,{x,xmin,xmax}] рисует график f как функции переменной x, изменяющейся от $xmin$ до $xmax$.

Plot3D[f,{x,xmin,xmax},{y,ymin,xmax}] рисует график f как функции переменных x и y, изменяющихся в пределах от $xmin$ до $xmax$ и от $ymin$ до $ymax$ соответственно.

Математический анализ

D[f,x] — частная производная f по переменной x. D[f,{x1,n1},{x2,n2},...] вычисляет смешанную частную производную от f по переменной $x1$ порядка $n1$, по переменной $x2$ порядка $n2$ и т.д.

Derivative[1][f][x], или **f′**, производная первого порядка от функции f одной переменной x. Используется при задании обыкновенных дифференциальных уравнений. Derivative[$n1$,$n2$,...][f][$x1$,$x2$,...] — полная

форма вычисленного выражения D[f[$x1$,$x2$,...],{$x1$,$n1$},{$x2$,$n2$}...] в случае, когда „Математика“ не может явно вычислить эту производную.

DSolve[ode, y[x], x] — решает обыкновенное дифференциальное уравнение *ode* с неизвестной функцией $y[x]$ от независимой переменной x. Если первый аргумент — список обыкновенных дифференциальных уравнений, а второй — список неизвестных функций, то решается система. Список *ode* может содержать данные Коши.

Dt[f, x] — полная производная от f по x, Dt[f] — дифференциал f. В случае задания Dt[f,$x1$,$x2$,...] последнее есть полная смешанная производная от f.

Integrate[f, x] — неопределенный интеграл от f по x. Если второй аргумент задан в виде $\{x, a, b\}$, то рассматриваемое выражение есть определенный интеграл от f по x в пределах от a до b. Integrate[f, $\{x, xmin, xmax\}, \{y, ymin, ymax\}, \ldots$] — повторный интеграл.

InverseSeries[s] — отрезок ряда Тэйлора для функции, обратной к функции f, отрезком ряда Тэйлора которой является s.

Product[f, {i, imin, imax}] — произведение f по i от $imin$ до $imax$. По умолчанию $imin = 1$.

Residue[expr, {x, x0}] — вычет *expr*, рассматриваемого как функция переменной x в точке $x0$.

Series[f, {x, x0, n}] — отрезок ряда Тэйлора f по переменной x в окрестности $x0$, содержащий степени до $(x - x0)^n$ включительно. Имеет заголовок SeriesData.

SeriesData[x, x0, {a0, a1, ...}, nmin, nmax, den] — выражение „Математики“ для представления суммы степеней вида $(x - x0)^{(nmin+i)/den}$ по i от 0 до $(nmax - nmin)/den$ с коэффициентами ai.

Sum[f, i, imin, imax] — сумма f по i от $imin$ до $imax$. По умолчанию $imin = 1$.

Предикаты и логические операции

And[p, q, ...], или **p && q && ...**, — логическая функция „И“ (конъюнкция).

AtomQ[expr] принимает значение True, если вычисленное *expr* есть атомарное выражение, False — в остальных случаях.

DigitQ[string] принимает значение True, если все знаки в строке *string* цифры, False — в остальных случаях.

EvenQ[expr] принимает значение True, если вычисленное *expr* есть четное целое число, False — в остальных случаях.

Equal[lhs, rhs], или **lhs == rhs**, принимает значение True, если *lhs* и *rhs* идентичны.

FreeQ[expr, form] принимает значение True, если ни одно из подвыражений *expr* не содержится в классе выражений „Математики", определяемом шаблоном *form*, и False — в остальных случаях.

Greater[x, y], или **x > y**, принимает значение True, если x численно строго больше y.

GreaterEqual[x, y], или **x >= y**, принимает значение True, если x численно больше или равно y.

Implies[p, q] — логическая импликация, принимающая значение False, если p вычисляется на True, а q на False.

IntegerQ[expr] принимает значение True, если *expr* при вычислении дает целое число, False — в остальных случаях.

Less[x, y], или **x < y**, принимает значение True, если x численно строго меньше y.

LessEqual[x, y], или **x <= y**, принимает значение True, если x меньше или равно y.

ListQ[expr] принимает значение True, если *expr* после вычисления есть список, False — в остальных случаях.

LogicalExpand[expr] раскрывает скобки в выражениях, содержащих логические операции конъюнкции && и дизъюнкции ||, используя дистрибутивность && относительно ||.

MatchQ[expr, form] определяет, принадлежит ли выражение *expr* классу выражений „Математики", задаваемому шаблоном *form*. Если принадлежит, то имеет значение True, если нет — False.

MemberQ[list, form] принимает значение True, если какой-либо элемент *list* соответствует *form*.

Negative[x] принимает значение True, если x есть отрицательное число.

NonNegative[x] принимает значение True, если x есть неотрицательное число.

Not — логическое отрицание.

NumberQ[expr] принимает значение True, если вычисленное *expr* есть число, False в остальных случаях.

OddQ[expr] принимает значение True, если вычисленное *expr* есть нечетное целое число, False в остальных случаях.

Or[p, q, ...], или **p || q || ...** — логическое неисключительное „ИЛИ" (дизъюнкция).

OrderedQ[h[x1, x2, ...]] принимает значение True, если элементы $x1$, $x2$ и т.д. канонически упорядочены, и False — в противном случае.

PolynomialQ[expr, vars] принимает значение True, если вычисленное *expr* есть многочлен по переменным *var*, и False — в противном случае.

Positive[x] принимает значение True, если *x* есть положительное число.

PrimeQ[x] принимает значение True, если *x* есть простое число.

SameQ[lhs, rhs], или **lhs === rhs**, принимает значение True, если *lhs* и *rhs* идентичны, и значение False — в противном случае.

StringQ[expr] принимает значение True, если вычисленное *expr* есть строка, и False в противном случае.

TrueQ[expr] принимает значение True, если вычисленное *expr* имеет значение True, и False — в противном случае.

Unequal[lhs, rhs], или **lhs =! = rhs**, принимает значение False, если *lhs* и *rhs* не идентичны.

UnsameQ принимает значение True, если вычисленное *expr* имеет значение True, и False — в противном случае.

VectorQ[expr] принимает значение True, если вычисленное *expr* есть одноуровневый список, и False — в противном случае.

Xor[p, q, ...] — логическое исключительное „ИЛИ“.

Списки

Порождение списков

Array[f, n] — список длины *n* с элементами $f[i]$. Array$[f, \{n_1, n_2, \ldots\}]$ — вложенный список с элементами $f[i_1, i_2, \ldots]$. Array[*f*, *dims*, *origin*] — список, значения индексов в котором начинаются с *origin* (по умолчанию $origin = 1$). Array[*f*, *dims*, *origin*, *h*] использует заголовок *h* вместо заголовка List на каждом уровне списка.

DiagonalMatrix[list] — диагональная матрица с элементами из списка *list* на главной диагонали.

IdentityMatrix[n] — единичная $n \times n$ матрица.

Range[n] — список $\{1, 2, \ldots, n\}$, Range$[m, n]$ — список $\{m, m+1, \ldots, m+ki\}$, в котором $m + ki < n < m + (k+1)i$. Числа m, n, i не обязательно целые.

Table[expr, n] — список из *n* копий выражения *expr*, а Table[*expr*, $\{i, n\}$] — список значений выражения *expr*, зависящего от *i*, для *i* от 1 до *n*. Список Table[*expr*, $\{i, m, n\}$] начинается с $i = m$. При вычислении Table[*expr*, $\{i, m, n, di\}$] используется шаг *di*. Вычисление выражения Table[*expr*, $\{i, imin, imax\}, \{j, jmin, jmax\}, \ldots$] порождает вложенный список.

Визуализация списков

ColumnForm[list] печатает столбец, в котором первый элемент списка *list* стоит наверху, ниже идет второй элемент и т.д. Задание второго аргумента, принимающего значения: Center — в центре, Left — слева (по умолчанию), Right — справа определяет расположение элементов по горизонтали. Задание третьего аргумента: Above — нижний элемент списка на уровне базовой линии, Below — верхний элемент списка на уровне базовой линии (по умолчанию), Center — столбец центрирован по базовой линии — определяет расположение элементов по вертикали. Первый аргумент функции не обязательно имеет заголовок *List*.

ListPlot[{y1,y2,...}] представляет графически точки с координатами $\{i, y_i\}$, а ListPlot[$\{\{x_1,y_1\},\{x_2,y_2\},\ldots\}$] точки $\{x_i, y_i\}$ списка с x-координатой x_i и y-координатой y_i.

ListDensityPlot[array] — плотностный график по массиву высот *array*.

MatrixForm[list] печатает элементы списка *list* следующим образом. Каждый элемент списка заключен в квадратную ячейку одинакового с другими размера. Одноуровневый список печатается в виде столбца, а двухуровневый — в стандартной матричной форме. Списки с более глубокими уровнями печатаются по умолчанию с последовательными уровнями чередующимися между столбцами и строками.

TableForm[list] печатает элементы списка *list* в виде массива прямоугольных ячеек. Высота каждой строки и ширина каждого столбца определяются максимальными размерами элементов. В отличие от MatrixForm, ячейки, в которые заключены элементы, одного и того же размера. Линейный список печатается в виде столбца, двухуровневый — в виде двумерной таблицы.

Получение общей информации о списке

Dimensions[expr] дает список размерностей *expr*. Dimensions[*expr*, *n*] дает список размерностей до уровня *n*. Все части выражения *expr* на каждом уровне должны иметь одинаковую длину. Каждый уровень *expr* должен содержать одинаковые заголовки.

Length[expr] дает число элементов в *expr*. Для атомарных объектов это число равно нулю.

Position[expr, pattern] дает список позиций, на которых стоят объекты в *expr*, соответствующие шаблону *pattern*. Position[*expr*, *pattern*, *levelspec*] дает список объектов на уровнях, определенных спецификацией уровня *levelspec*. По умолчанию спецификация *levelspec* равна *Infinity* с опцией Heads → True. Position[*expr*, *pattern*, *levelspec*, *n*] дает позиции первых *n* частей *expr*, отвечающих шаблону.

Операции над списками

Append[list, elem] добавляет элемент *elem* последней по номеру частью *list*.

Complement[list, e_1, e_2, ...] дает список элементов в *list*, не содержащихся ни в одном из e_i.

Delete[list, n] удаляет элемент на позиции *n* в *list*. Delete[*list*, {*i*, *j*, ...}] удаляет часть *list* на позиции {*i*, *j*, ...}. Delete[*list*, {{i_1, j_1, ...}, {i_2, j_2, ...} ...}] удаляет части *list* на нескольких позициях. Удаление заголовка элемента эквивалентно применению функции **FlattenAt**.

Drop[list, n] дает *list* с удаленными *n* первыми (при отрицательном *n* последними) элементами. Drop[*list*, {*n*}] удаляет *n*-й элемент. Drop[*list*, {*m*, *n*}] удаляет с *m*-го по *n*-й элемент.

First[list] или **list[[1]]** извлекает первый элемент из *list*.

Flatten[list] приводит *list* к одному уровню. При задании второго аргумента *n* действует до уровня *n*. Flatten[*list*, *n*, *h*] действует на подвыражения с заголовком *h*.

FlattenAt[list, n] действует функцией Flatten на подсписок, являющийся *n*-м элементом списка *list*. FlattenAt[*expr*, {*i*, *j*, ...}] применяет Flatten к части выражения *expr* на позиции {*i*, *j*, ...}. FlattenAt[*expr*, {{i_1, j_1, ...}, {i_2, j_2, ...}, ...}] — то же, что и выше, на нескольких позициях.

Insert[list, elem, n] — элемент *elem* вставляется на позицию *n* в списке *list*. Если вместо *n* задан список {*i*, *j*, ...}, то *elem* вставляется на позицию {*i*, *j*, ...}. При задании третьим аргументом списка {{i_1, j_1, ...}, {i_2, j_2, ...}, ...} элемент *elem* вставляется на несколько позиций. Выражение *list* может иметь и иной, нежели List, заголовок.

Intersection[$list_1$, $list_2$, ...] *дает упорядоченный список элементов, общих* для всех $list_i$. Выражения $list_i$ должны иметь одинаковый заголовок, но не обязательно List.

Join[$list_1$, $list_2$, ...] дает конкатенацию списков $list_i$. Join можно применять к выражениям, имеющим одинаковый заголовок, отличный от List.

Last[list], или **list[[−1]]**, — последний элемент в *list*.

Part[list, i], или **list[[i]]**, дает *i*-ю часть выражения *list*, причем list[[0]] есть заголовок *list*. Part[*list*, *i*, *j*, ...] извлекает *i*-ю часть *list*, из нее *j*-ю часть и т.д. Part[*list*, {i_1, i_2, ...}] дает список частей i_1, i_2 и т.д. выражения *list*. Можно делать присвоения вида *t*[[i]] = *value*, чтобы изменять i-ю часть выражения *t*. В случае, если *list* есть список, *list*[[{i_1, i_2, ...}]] дает список частей. Если же *list* имеет иной заголовок, то он применяется к списку частей.

Partition[list, n] разбивает *list* на неперекрывающиеся части длины *n*. Задание третьим аргументом числа *d* порождает разбиение длины *n* с отступом *d*. Partition[*list*, $\{n_1, n_2, \ldots\}$, $\{d_1, d_2, \ldots\}$] разбивает последовательные уровни в выражении *list* на части длины n_i с отступом d_i. При разбиении на части длины *n* те элементы, стоящие в конце выражения *list*, которые не вошли в последнюю часть длины *n*, опускаются. Если $d > n$, то опускаются промежуточные между последовательными частями элементы. Объект *list* не обязательно имеет заголовок List.

Prepend[list, elem] дает *list* с добавлением *elem* в качестве первого элемента.

ReplacePart[list, new, n] приводит к выражению, в котором *n*-я часть *list* заменена на *new*. ReplacePart[*list*, *new*, $\{i, j, ..\}$] заменяет часть на позиции $\{i, j, \ldots\}$. ReplacePart[*list*, *new*, $\{\{i_1, j_1, \ldots\}, \{i_2, j_2, \ldots\}\}$] заменяет части на нескольких позициях.

Rest[list] дает *list* с удаленным первым элементом.

Reverse[list] обращает порядок частей *list*.

RotateLeft[list, n] циклически переставляет части выражения *list* на *n* позиций влево. RotateLeft[*list*] предполагает, что $n = 1$. RotateLeft[*list*, $\{n_1, n_2, \ldots\}$] циклически переставляет элементы на уровнях *i* на n_i позиций влево.

RotateRight аналогична RotateLeft с циклическими перестановками вправо.

Sort[list] сортирует элементы *list* в соответствии с каноническим порядком. Функцию порядка *p* можно задать вторым аргументом в **Sort**. Канонический порядок для строк определяется порядком символов в *$StringOrder*. Символы упорядочиваются в соответствии с их именами. Целые, рациональные и вещественные числа упорядочиваются по величине. Выражения упорядочиваются по глубине. Более короткие выражения идут раньше. Sort[*list*, *p*] применяет функцию *p* к парам элементов и переставляет элементы в паре в соответствии с *p*. По умолчанию *p* есть *OrderedQ*[{#1, #2}]&. Sort может применяться к любым выражениям.

Take[list, n] дает первые *n* элементов *list*. Take[*list*, $\{m, n\}$] дает элементы с *m*-го по *n*-й. Выражение *list* не обязательно список.

Transpose[list] переставляет первые два уровня в выражении *list*, а вычисление выражения Transpose[*list*, $\{n_1, n_2, \ldots\}$] перемещает *i*-й уровень на n_i-й уровень нового выражения.

Union[$list_1$, $list_2$, ...] дает упорядоченный список элементов, содержащихся хотя бы в одном из $list_i$.

Матрицы

CharacteristicPolynomial[m,x] вычисляет характеристический полином квадратной матрицы *m*, являющийся полиномом по переменной *x* степени Length[*m*].

Det[m] вычисляет детерминант квадратной матрицы *m*.

Dot[a,b,c], или *a.b.c*, вычисляет произведения векторов, матриц и тензоров. Выражение *a.b* имеет смысл, если *a* и *b* есть списки подходящих размерностей. При этом происходит свертывание последнего индекса *a* с первым индексом *b*. Результатом применения функции Dot к тензорам $T_{i_1,i_2,\ldots,i_n}$ и $U_{j_1,j_2,\ldots,j_m}$ является тензор $\sum_k T_{i_1,\ldots,i_{n-1},k}U_{k,j_2,\ldots,j_m}$. В случае, если аргументами функции Dot являются не списки, то Dot не вычисляется. Dot имеет атрибут Flat.

Eigenvalues[m] — список собственных значений квадратной матрицы *m*.

Eigenvectors[m] — список собственных векторов квадратной матрицы *m*.

Eigensystem[m] — список {*values*,*vectors*} собственных значений и собственных векторов квадратной матрицы *m*. Эта функция находит численные значения этих объектов, если *m* содержит вещественные числа. Элементы *m* могут быть комплексными. Все собственные векторы линейно независимы. Если число собственных векторов равно числу ненулевых собственных значений, то векторы и значения даны в соответствующих позициях в их списках. Если же собственных значений больше, чем независимых векторов, то каждое лишнее собственное значение спаривается с нулевым вектором.

MatrixPower[m,n] дает *n*-ю степень квадратной матрицы *m* для положительных и отрицательных *n*.

Minors[m,k] дает матрицу, состоящую из детерминантов $k \times k$ подматриц матрицы *m*. Детерминанты располагаются в лексикографическом порядке.

Transpose[m] — транспонированная с *m* матрица.

Проверка свойств и поиск элементов списка

Count[list,pattern] — число элементов в выражении *list*, отвечающих *pattern*. Count[*expr*,*pattern*,*levelspec*] дает общее число подвыражений в *expr*, отвечающих *pattern*, на уровнях, задаваемых *levelspec*.

FreeQ[expr,form] имеет значение True, если никакое подвыражение выражения *expr* не соответствует *form*, и имеет значение False — в противном случае. Выражение *form* может быть шаблоном. FreeQ[*expr*,*form*,*levelspec*] проверяет только те уровни, которые определены спецификацией уровня *levelspec*.

MemberQ[list, form] имеет значение True, если какой-либо элемент списка *list* отвечает *form*, и False — в противном случае. MemberQ[*list*, *form*, *levelspec*] проверяет части списка, заданные *levelspec*. Выражение *form* может быть шаблоном.

Применение заголовков функций к спискам

Apply[f, expr], или *f*@@*expr*, заменяет заголовок *expr* на *f*; при задании третьим аргументом *levelspec* заголовки будут заменяться в частях *expr*, определяемых *levelspec*.

Inner[f, list₁, list₂, g] есть обобщение функции **Dot**, причем *f* играет роль умножения, а *g* сложения. По умолчанию *g* принимает значение Plus. Как и Dot, Inner свертывает последний индекс первого тензора с первым индексом второго тензора. Применение Inner к тензорам рангов r и s соответственно приводит к тензору ранга $r+s-2$. Inner[f, $list_1$, $list_2$, g, n] свертывает n-й индекс первого тензора с первым индексом второго. Заголовки $list_1$ и $list_2$ должны быть одинаковыми, но не обязательно List.

Map[f, expr], или *f* /@*expr*, применяет *f* к каждому элементу на первом уровне *expr*, а Map[*f*, *expr*, *levelspec*] применяет *f* к частям *expr*, определяемых *levelspec*.

MapAll[f, expr], или *f* //@*expr*, применяет *f* к каждому подвыражению *expr*, что эквивалентно Map[*f*, *expr*, {0, *Infinity*}].

MapAt[f, expr, n] применяет *f* к элементу в позиции *n* в *expr*. MapAt[*f*, *expr*, {*i*, *j*, ...}] применяет *f* к части *expr* на позиции {*i*, *j*, ...}; MapAt[f, $expr$, $\{\{i_1, j_1, \ldots\}, \{i_2, j_2, \ldots\}, \ldots\}$] применяет *f* к частям *expr* на нескольких позициях.

MapIndexed[f, expr] применяет *f* к элементам *expr*, порождая спецификацию каждого элемента как второго аргумента *f*; MapIndexed[*f*, *expr*, *levelspec*] применяет *f* ко всем частям *expr* на уровнях, задаваемых *levelspec*.

MapThread[f, {{a₁, a₂, ...}, {b₁, b₂, ...}, ...}] имеет в качестве результата выражение $\{f[a_1, b_1, \ldots], f[a_2, b_2, \ldots], \ldots\}$; MapThread[$f$, $\{expr_1, expr_2, \ldots\}$, n] применяет *f* к частям $expr_i$ на уровне *n*.

Outer[f, list₁, list₂, ...] — обобщенное тензорное произведение аргументов $list_i$, понимаемых как тензоры соответствующего ранга. Outer[Times, $list_1$, $list_2$, ...] — обычное тензорное произведение. Результатом применения Outer к тензорам $T_{i_1,i_2,\ldots,i_r}$ и $U_{j_1,j_2,\ldots,j_s}$ является тензор $V_{i_1,i_2,\ldots,i_r,j_1,\ldots,j_s}$ с элементами $f[T_{i_1,\ldots,i_r}, U_{j_1,\ldots,j_s}]$. Заголовки $list_i$ должны совпадать, но не обязаны быть List.

Thread[f[args]] вычисляет *f*[*args*] и с заголовком вычисленного выражения работает как **MapThread**. Thread[*f*[*args*],*h*] продевает *f* сквозь все объекты в *args*, имеющие заголовок *h*. Thread[*f*[*args*], *h*,*n*] продевает *f* сквозь все объекты с заголовком *h* в первых *n* *args*; Thread[*f*[*args*],*h*,−*n*] продевает сквозь последние *n* *args*; Thread[*f*[*args*],*h*,{*m*,*n*}] продевает в аргументах с номерами от *m* до *n*. Функции, имеющие атрибут *Listable*, автоматически продеваются сквозь списки. Все элементы в выделенных *args* с заголовком *h* должны иметь одинаковую длину. Аргументы, не имеющие заголовка *h*, копируются столько раз, какова длина элементов с заголовком *h*.

Строки

StringDrop["string",n] удаляет первые *n* знаков строки.

StringInsert["string","new",n] вставляет знак *new* на *n*-е место в строке.

StringJoin["string1","string2",...] имеет значением строку, являющуюся конкатенацией строк *string*1, *string*2 и т.д.

StringLength["string"] — число знаков в строке.

StringMatchQ["string","pattern"] имеет значение True, если строка *string* соответствует строке *pattern*, где *pattern* может содержать специальные символы * (0 или более знаков), @ (0 или более строчных букв), * (сам знак *).

StringPosition["string","somth"] имеет значением список пар чисел, первое из которых — позиция начала строки *somth* в строке *string*, второе — позиция конца *somth*.

StringQ[expr] принимает значение True, если вычисленное *expr* есть строка, и False — в остальных случаях.

StringReplace["string","s1" → "s2"] заменяет часть *s*1 строки *string* на *s*2.

StringReverse["string"] обращает строку.

StringTake["string",n] имеет значением строку, состоящую из первых *n* знаков строки *string*.

Уравнения

Представление уравнений и их решений

And[e_1, e_2, ...], или e_1 && e_2 && ..., есть логическая функция „И“. Функция And используется для представления системы уравнений в виде $lhs_1 == rhs_1$ && $lhs_2 == rhs_2$ && ...

Equal[lhs, rhs], или lhs == rhs, дает результат True, если *lhs* и *rhs* совпадают как выражения „Математики", и False, если *lhs* и *rhs* явно не идентичны, что устанавливается, например, при сравнении чисел или строк. В остальных случаях Equal не вычисляется. Функция Equal используется для представления уравнений и систем уравнений в символьной форме. Возможна запись Equal[e], что вычисляется как True.

Or[e_1, e_2, ...], или e_1 || e_2 || ..., есть логическая функция „ИЛИ". Используется при записи решений.

Алгебраические и трансцендентные уравнения

AlgebraicRules[eqns, {x_1, x_2, ...}] порождает список алгебраических правил, по которым переменные, стоящие ранее в списке,заменяются на переменные, стоящие позже, в соответствии с уравнениями *eqns*. AlgebraicRules порождает объект AlgebraicRulesData, который содержит представление в виде базиса Гребнера для уравнений *eqns*.

Eliminate[eqns, vars] исключает переменные *vars* в системе уравнений *eqns*. Переменные могут быть любыми выражениями. Уравнения должны быть представлены в форме *lhs* == *rhs* и объединены в систему либо с помощью списка, либо с помощью &&. Функция Eliminate предназначена главным образом для работы с линейными и полиномиальными уравнениями.

GroebnerBasis[{$poly_1$, $poly_2$, ...}, {x_1, x_2, ...}] вычисляет список полиномов, которые образуют один из возможных базисов Гребнера идеала, порожденного $poly_i$. Базис Гребнера может зависеть от порядка, в котором даны x_i.

NSolve[lhs == rhs, var] дает список приближенных численных значений корней полиномиального уравнения. NSolve[*eqn*, *var*, *n*] дает результат с *n* знаками после запятой. NSolve[*eqn*, *var*] эквивалентно N[Solve[*eqn*, *var*]], за исключением точности вычислений.

Reduce[eqns, vars] упрощает уравнения *eqns* с тем, чтобы решить их относительно *vars*. Уравнения, порождаемые Reduce, эквивалентны *eqns* и содержат все их возможные решения. Reduce[*eqns*, *vars*, *elims*] упрощает уравнения, стараясь исключить переменные *elims*. Уравнения должны быть заданы в форме *lhs* == *rhs*, а системы заданы с помощью списка или &&. Функция Reduce предназначена в основном для решения полиномиальных уравнений.

Roots[lhs == rhs] приводит к дизъюнкции уравнений, которая представляет корни полиномиального уравнения. Roots использует функции Factor и Decompose для нахождения корней.

Solve[eqns, vars] решает уравнение или систему уравнений относительно переменных *vars*. Solve[*eqns*, *vars*, *elims*] решает уравнения относительно *vars*, исключая переменные *elims*. Уравнения должны быть представлены в форме *lhs* == *rhs*, а система уравнений с помощью списка или булевых операций &&. Solve[*eqns*] решает уравнения относительно всех входящих переменных. При наличии нескольких решений ответ представляется в виде списка. Кратные решения представляются столько раз, какова их кратность. Solve предназначается главным образом для решения линейных и полиномиальных уравнений. Опция InversFunctions определяет, должны ли использоваться обратные функции для представления решений более общих уравнений. По умолчанию опция InverseFunction установлена на Automatic. В этом случае Solve пользуется обратными функциями, но печатает соответствующее предупреждение. Solve ищет общие решения, игнорируя частные случаи, возникающие при специальных значениях параметров, входящих в уравнения. Полное исследование таких случаев проводит функция Reduce. Solve не всегда получает все решения. Эта функция приводит явные решения, которые ей удалось найти, и дает символическое представление оставшихся ненайденными решений в терминах функции Roots. Solve дает ответ {}, если уравнения не имеют решений. Solve[*eqns*, Mode → Modular] решает уравнения, трактуя равенства по модулю целого числа,которое можно задать, присоединяя уравнение Modulus == *p*. Если такого уравнения нет, Solve пытается найти решения для возможных целых чисел.

SolveAlways[eqns, vars] дает значения параметров, таких, что уравнения справедливы при любых значениях переменных *vars*. SolveAlways предназначена в основном для линейных и полиномиальных уравнений. Она получает соотношения между параметрами, явно входящими в уравнения, но не входящими в список *vars*. SolveAlways[*eqns*, *vars*] эквивалентно Solve[!Eliminate[!*eqns*, *vars*]].

ToRules[eqns] применяется к логической комбинации уравнений в форме, порождаемой функциями Roots и Reduce и преобразует их в список правил в форме, генерируемой Solve. ToRules игнорирует неравенства (! =) и, следовательно, дает только „общие“ решения.

Числа, функции с численными аргументами и численные методы

Abs[z] — абсолютное значение числа z.

ArcCos[z] — значение функции *Arccos* комплексного аргумента z. Для вещественного аргумента изменяется от 0 до π.

ArcCosh[z] — значение обратной функции к гиперболическому косинусу от комплексного числа z.

ArcCot[z] — значение функции *Arcctg* от комплексного числа z.

ArcSin[z] — значение функции *Arcsin* комплексного аргумента z. Для вещественного аргумента изменяется от $-\pi/2$ до $\pi/2$.

ArcSinh[z] — значение обратной к гиперболическому синусу функции комплексного аргумента z.

ArcTan[z] — арктангенс комплексного числа z. Значения заключены между $-\pi/2$ и $\pi/2$.

ArcTanh[z] — значение обратной к гиперболическому тангенсу функции от комплексного аргумента z.

Arg[z] — аргумент комплексного числа z. Изменяется от $-\pi$ до π.

Beta[a, b] — Эйлерова бета-функция.

Binomial[n, m] — биномиальный коэффициент.

Ceiling[x] — наименьшее целое, превосходящее вещественное число x. Для комплексного числа $a + Ib$ с a и b типа Real имеет результатом комплексное число Ceiling[a] + I Ceiling[b].

Chop[expr] заменяет вещественные числа, меньшие 10^{-10}, в выражении *expr* на 0. Chop[*expr*, *eps*] заменяет на 0 вещественные числа по абсолютной величине меньшие *eps*.

Conjugate[z] — комплексное число, сопряженное с z.

Cos[z] — косинус комплексного числа z.

Cosh[z] — гиперболический косинус комплексного числа z.

Cot[z] — котангенс комплексного аргумента z.

Degree — число радиан в одном градусе. Применяется в аргументах тригонометрических функций.

Divide[x, y] — то же самое, что xy^{-1}.

Divisors[n] — список делителей целого числа n.

E — основание натуральных логарифмов.

Erf[z] — функция ошибок erf.

EulerGamma — эйлерова константа гамма.

EvenQ[n] — одноместный предикат, принимающий значение True, если n — четное целое число, и значение False — в противном случае.

Exp[z], или E^x, — экспоненциальная функция.

Factorial[n], или n!, — факториал целого числа n. Для нецелых n определяется как Эйлерова функция $\Gamma[n+1]$.

FactorInteger[n] — список простых делителей целого числа n вместе с их степенями.

FindMinimum[f, {x, x0}] ищет локальный минимум функции $f(x)$ в окрестности точки $x0$.

FindRoot[lhs == rhs, {x, x0}] ищет численное решение уравнения $lhs == rhs$ в окрестности точки $x0$.

Fit[data, funs, vars] имеет значением наилучшее среднеквадратичное приближение дискретных данных *data* с помощью суммы функций *funs*.

Floor[x] — наибольшее целое, не превосходящее вещественное число x.

Fourier[data] — дискретное преобразование Фурье.

Gamma[z] — Эйлерова гамма-функция.

GCD[n_1, n_2, ...] — наибольший общий делитель чисел n_1, n_2 и т.д.

I — мнимая единица.

Im[z] — мнимая часть комплексного числа z.

Infinity — символ, представляющий положительную бесконечность.

IntegerDigits[n] — список цифр в десятичном представлении целого числа n. Задание второго аргумента b приводит к списку цифр в представлении числа n в системе исчисления с базой b.

IntegerQ — одноместный предикат, имеющий значение True, если аргумент — целое число, и False — в остальных случаях.

InterpolatingPolynomial[{f_1, f_2, ...}, var] вычисляет полином по переменной *var*, совпадающий с f_i в точке i. InterpolatingPolynomial[{{x_1, f_1}}, {x_2, f_2}, ...}, *var*] вычисляет полином по переменной *var*, совпадающий с f_i в точке x_i.

InverseFourier[data] — преобразование, обратное к дискретному преобразованию Фурье.

LCM[n, m] — наименьшее общее кратное целых чисел n и m.

Log[z] — логарифм по натуральному основанию комплексного числа z.

Max[x1, x2, ...] — максимальное из чисел $x1$, $x2$ и т.д.

Min[x1, x2, ...] — минимальное из чисел $x1$, $x2$ и т.д.

Mod[m, n] — остаток от деления целого числа m на целое n.

N[expr] — представление *expr* в виде вещественного числа. Задание второго аргумента n определяет вычисление *expr* с n значащими цифрами.

NDSolve[ode, y, {x, xmin, xmax}] дает решение задачи Коши для одного дифференциального уравнения или системы дифференциальных уравнений. Аргумент *ode* включает как сами дифференциальные уравнения, заданные с помощью функции Equal, или ==, так и начальные условия, поставленные на концах интервала интегрирования *xmin* или *xmax*. Второй аргумент *y* есть список заголовков неизвестных функций, *x* — независимая переменная.

Negative[x] принимает значение True, если *x* есть отрицательное число, False, если *x* есть неотрицательное число, и не определено в остальных случаях.

NIntegrate[f, {x, xmin, xmax}] дает численное приближение для определенного интеграла от *f* по переменной *x* на отрезке от *xmin* до *xmax*.

NonNegative[x] принимает значение True, если *x* есть неотрицательное число, False, если *x* есть отрицательное число, и не определено в остальных случаях.

NRoots[lhs == rhs, var] — список численных значений для корней полиномиального уравнения *lhs == rhs* относительно неизвестной *var*.

NSolve[eqns, vars] численно решает уравнение или систему уравнений относительно неизвестных *vars*. Задание третьего аргумента *elims* определяет список исключаемых при решении неизвестных.

NumberQ — предикат, принимающий значение True, если его аргумент — число любого типа, и False — в других случаях.

OddQ — предикат, принимающий значение True, если его аргумент — нечетное целое число, и False — в других случаях.

Pi — число π.

Plus[x, y, ...] — сумма *x*, *y* и т.д.

Positive[x] принимает значение True, если *x* неотрицательное число, False, если *x* отрицательное число, не определено в остальных случаях.

Prime[n] — *n*-е по счету простое число.

PrimePi[x] — количество простых чисел, меньших или равных числу *x*.

PrimeQ — предикат, принимающий значение True, если его аргумент — простое число, False — в остальных случаях.

Quotient[m, n] — частное от деления целого числа *m* на целое *n*.

Random при последовательном вычислении дает равномерно распределенные псевдослучайные вещественные числа на отрезке от 0 до 1. Random[*type*, *interval*] при последовательном вычислении дает равномерно распределенные псевдослучайные числа типа *type* (вещественные, целые или комплексные), заключенные в области *interval*. В случае комплексных чисел область задается парой комплексных чисел.

Rationalize[x] — рациональное приближение вещественного числа x. Второй аргумент может задавать точность приближения.

Re[z] — вещественная часть комплексного числа z.

Round[x] округляет вещественное число x до ближайшего целого.

Sin[z] — синус комплексного числа z.

Sinh[z] — гиперболический синус комплексного числа z.

Tan[z] — тангенс комплексного числа z.

Tanh[z] — гиперболический тангенс комплексного числа z.

Times[x, y, ...] — произведение x, y и т.д.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

A

Abs 248
AbsoluteThickness 93
AbsolutePointSize 93
Accuracy 65
AiryAi 96
AiryAiPrime 30
AlgebraicRules 246
AmbientLight 84
And 44, 237
Apart 41, 232
Append 103, 241
Apply 31, 244
ArcCos 248
ArcCot 248
ArcSin 248
ArcTan 248
ArcTanh 248
Arg 248
Array 55, 103, 239
AspectRatio 71
AtomQ 17, 237
Attributes 119
Axes 71
AxesLabel 71
AxesOrigin 77
AxesStyle 72

B

Background 72
BarChart 91
Begin 199
BeginPackage 201
BesselJ 23
Beta 248
Binomial 248
Blank 127
BlankNullSequence 149
BlankSequence 149
Boxed 83
BoxRatios 85
BoxStyle 84

C

Cancel 41, 232
Cases 151
Ceiling 67, 248
CForm 216
Character 213
CharacteristicPolynomial 112, 243
ChebyshevT 78
Chop 67, 248
Circle 94
Clear 38
ClearAttributes 181
ClipFill 85
Close 216
Coefficient 36, 232
CoefficientList 36, 232
Collect 36, 232
ColorFunction 85
ColorOutput 73
ColumnForm 240
Complement 107, 241
ComplexExpand 233
ComposeList 135
Composition 134
Compiled 79
CompoundExpression 165

Condition 152
Cone 98
Conjugate 248
Constant 184
Context 198
ContourGraphics 73
ContourLines 82
ContourPlot 30, 81, 235
ContourPlot3D 89
Contours 81
ContourShading 82
ContourSmoothing 82
ContourStyle 81
Cos 248
Cosh 248
Cot 248
Count 107, 243
Cuboid 96
Cylinder 98

D

D 23, 51, 236
Dashing 72
Decompose 233
DefaultColor 73
DefaultFont 73
Degree 248
Delete 16, 105, 241
DeleteCases 151
Denominator 42, 233
DensityGraphics 73
DensityPlot 82, 235
Depth 115
Derivative 52, 236
Det 111, 243
DiagonalMatrix 103, 239
DigitQ 237
Dimensions 240
Disk 95
DisplayFunction 73
Divide 248
Divisors 138, 248
Do 165
Dot 111, 243
DoubleHelix 98
DownValues 142, 178
Drop 105, 241
DSolve 24, 58, 237
Dt 52, 237

E

E 33, 248
Eigensystem 112, 243
Eigenvalues 112, 243
Eigenvectors 112, 243
Eliminate 48, 246
End 203
EndPackage 203
Epilog 95
Erf 248
Equal 18, 43, 238
EulerGamma 248
Evaluate 190
EvenQ 42, 237
Exp 15, 249
Expand 34, 233
ExpandAll 233
ExpandDenominator 42, 233
ExpandNumerator 42, 233
Exponent 234

F

Factor 11, 35, 234
Factorial 249
FactorInteger 249
FactorList 35, 234
FactorSquareFree 234
FactorSqyareFreeList 234
FactorTerms 35, 234
FactorTermsList 35, 234
FilledPlot 90
FindMinimum 249
FindRoot 23, 249
First 106, 241
Fit 28, 249

FixedPoint 133
Flat 182
Flatten 108, 241
FlattenAt 109, 241
Floor 67, 249
Fold 134
FoldList 134
FontForm 96
For 172
Format 220
FormatType 219
FormatValues 220
FortranForm 216
Fourier 249
Frame 73
FrameLabel 73
FrameStyle 74
FrameTicks 74
FreeQ 238
FullForm 114
Function 129

G

Gamma 249
GCD 249
Get 56, 210
GoldenRatio 71
Graphics 22, 73, 92
Graphics3D 96
GraphicsArray 87
GrayLevel 72
Greater 43, 238
GreaterEqual 43, 238
GridLines 78
GroebnerBasis 246

H

Head 17, 114
HeldPart 190
Helix 98
HiddenSurface 85
Hold 114, 187
HoldAll 170, 188
HoldFirst 188
HoldForm 114, 188
HoldRest 188
Hue 72

I

I 19, 249
Identity 135
IdentityMatrix 103, 239
If 167
Im 249
ImplicitPlot 57
Implies 238
In 9
Infinity 54, 249
Information 184
Inner 122, 244
InputForm 219
InputStream 214
Insert 104, 241
Integer 16
IntegerDigits 139, 249
IntegerQ 42, 238
Integrate 23, 53, 237
InterpolatingFunction 26
InterpolatingPolynomial 249
Intersection 107, 241
Inverse 111
InverseFourier 249
InverseFunction 135
InverseSeries 55, 237

J

Join 110, 241

L

Last 11, 106, 241
LCM 249
Length 31, 37, 114, 240
Less 43, 238
LessEqual 43, 238
Level 115

Lighting 84
LightSources 84
Line 92
LinearSolve 50
List 16, 100
Listable 119
ListContourPlot 83, 235
ListContourPlot3D 89
ListDensityPlot 83, 236
ListFilledPlot 90
ListPlot 28, 80, 236
ListPlot3D 83, 86, 236
ListQ 238
Literal 193
Locked 184
Log 11, 249
LogicalExpand 63, 238

M

Map 118, 244
MapAll 244
MapAt 119, 244
MapIndexed 244
MapThread 120, 244
MatchQ 149, 238
MathLink 4
MatrixForm 12, 50, 240
MatrixPower 112, 243
Max 249
MaxBend 79
MemberQ 238
Mesh 83, 85
Min 249
Minors 111, 243
Mod 249
Module 174
MoebiusStrip 98

N

N 20, 65, 249
Names 57
NDSolve 26, 61, 250
Negative 238
Needs 56
Nest 132
NestList 134
NIntegrate 69, 250
NonNegative 238
Normal 55
Not 44, 238
NRoots 250
NSolve 246
Null 168
Number 211
NumberQ 42, 238
Numerator 42, 234

O

OddQ 42, 238
OneIdentity 183
OpenRead 214
OpenWrite 215
Options 76
Or 44, 238
OrderedQ 238
Orderless 181
Out 9
Outer 122, 244
OutputForm 219
OutputStream 215
OwnValues 178

P

ParametricPlot 26, 79, 236
ParametricPlot3D 83, 85, 236
Part 17, 105, 114, 241
Partition 108, 242
Pi 27, 250
PieChart 91
Plot 21, 76, 236
Plot3D 30, 83, 236
PlotDivision 79
PlotJoined 80
PlotLabel 72
PlotPoints 79

PlotRange 75
PlotRegion 75
PlotStyle 78
Plus 16, 250
Point 92
PointSize 80
Polygon 94
PolynomialGCD 37, 234
PolynomialLCM 37, 234
PolynomialMod 235
PolynomialQ 37, 235, 239
PolynomialQuotient 37, 235
PolynomialRemainder 37, 235
Position 115, 240
Positive 43, 239
Power 16
PowerExpand 40, 235
Precision 65
Prepend 103, 242
Prime 250
PrimePi 250
PrimeQ 42, 239
Print 166
Product 167, 237
Prolog 95
Protect 144
Protected 142
Put 214
PutAppend 215

Q

Quotient 250

R

Random 101, 250
Range 101, 239
Rationalize 67, 251
Re 251
Read 214
ReadList 28, 211
ReadProtected 184
RecordLists 211
Rectangle 94
Reduce 50, 246
ReleaseHold 188
Remove 204
ReplaceAll 39
ReplaceHeldPart 190
ReplacePart 104, 242
ReplaceRepeated 40
Residue 237
Rest 105, 242
Resultant 37, 235
Reverse 106, 242
RGBColor 72
Roots 47, 247
RotateLabel 73
RotateLeft 107, 242
RotateRight 107, 242
Round 67, 251
Rule 19, 39
RuleDelayed 39, 147
RungeKutta 69

S

SameQ 44, 239
Save 215
Select 106
Sequence 137
SequenceForm 220
Series 27, 53, 237
SeriesData 55, 237
Set 38, 141
SetAttributes 180
SetDelayed 143
SetOptions 219
Shading 84
Show 29, 87
Simplify 24, 42, 235
Sin 15, 251
Sinh 251
Skip 214
Slot 130
Solve 18, 45, 247
SolveAlways 247
Sort 106, 242

Sphere 98
SphericalRegion 89
Splice 217
Sqrt 15
String 17
StringDrop 245
StringForm 220
StringInsert 245
StringJoin 245
StringLength 245
StringMatchQ 245
StringPosition 245
StringQ 239, 245
StringReplace 245
StringReverse 245
StringTake 245
Subscript 222
Subscripted 221
SubValues 179
Sum 167, 238
Superscript 222
SurfaceGraphics 92
Switch 170
Symbol 17

T

Table 55, 101, 239
TableForm 240
TableSpacing 216
Take 106, 242
Tan 251
Tanh 251
TeXForm 216
Text 95
Thickness 72
Thread 120, 245
Through 136
Ticks 75
Times 16, 251
Timing 33
ToExpression 213
Together 41, 235
ToRules 48, 248
Torus 98
ToString 216
Trace 17
TracePrint 17
Transpose 109, 243
Trig 35
True 42
TrueQ 239

U

Unequal 43, 239
Union 107, 243
UnsameQ 239
Unprotect 144
Unset 38
UpSet 145
UpSetDelayed 145
UpValues 145, 178

V

Variables 37, 235
VectorQ 239
ViewCenter 85
ViewPoint 85
ViewVertical 85

W

Which 169
While 171
Word 212
WriteString 215

X

Xor 239

$

$Context 198
$ContextPath 198
$MachinePrecision 66
$Packages 204

ЛИТЕРАТУРА

1. *Аладьев В.З., Хунт Ю.Я., Шишаков М.Л.* Математика на персональном компьютере. – Гомель. Российская академия космонавтики. Российская академия ноосферы, 1996.

2. *Wolfram S.* "Mathematica". A System for Doing Mathematics by Computer. Second Edition. – Addison-Wesley Publishing Company, 1991.

3. *Gray J.W.* Mastering Mathematica. Programming Methods and Applications. – AP Professional, 1994.

4. *Maeder R.E.* Programming in Mathematica. Second Edition. – Addison-Wesley Publishing Company, 1991.

5. *Maeder R.E.* The Mathematica Programmer. – AP Professional, 1994.

6. *Vvedenskii D.* Partial Differential Equations with "Mathematica". – Addison-Wesley Publishing Company, 1993.

7. *Gaylord R.J., Kamin S.N., Wellin P.R.* Introduction to Programming with Mathematica. – Springer-Verlag, 1993.

8. *Dzhamay A.V., Foursov M.V., Grishin O.I., Vorob'ev E.M.* and *Zhikharev V.N.* A "Mathematica" program SYMMAN for symmetry analysis of (overdetermined) systems of partial differential equations // New Computer Technologies in Control Systems, Proceedings of the International Workshop, Pereslavl-Zalessky. – Russia, 1994.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ 3

Часть I. „МАТЕМАТИКА“ КАК СИМВОЛЬНЫЙ, ГРАФИЧЕСКИЙ И ЧИСЛЕННЫЙ КАЛЬКУЛЯТОР 7

Глава 1. АЗБУКА „МАТЕМАТИКИ“ 8
1.1. Первый сеанс 8
1.2. Основы синтаксиса „Математики“ 14
1.3. Обзор „Математики“ 18
Упражнения 33

Глава 2. СИМВОЛЬНЫЕ ВЫЧИСЛЕНИЯ 34
2.1. Преобразования многочленов 34
2.2. Подстановки 38
2.3. Преобразования рациональных выражений 40
2.4. Предикаты и булевы операции 42
2.5. Алгебраические и трансцендентные уравнения 45
2.6. Математический анализ 51
2.7. Специализированные программы 56
2.8. Обыкновенные дифференциальные уравнения 58
2.9. Числа и операции над числами 64
Упражнения 68

Глава 3. ВСТРОЕННАЯ ГРАФИКА 70
3.1. Графические функции и их опции 70
3.2. Двумерная графика 76
3.3. Трехмерная графика 83
3.4. Изменение стиля и комбинирование построенных рисунков 87
3.5. Мультипликация 88
3.6. Графические функции специализированных пакетов 89
3.7. Графические примитивы 92
Упражнения 98

Глава 4. РАБОТА СО СПИСКАМИ 100
4.1. Порождение списков 100
4.2. Преобразования списков 103
4.3. Работа с векторами и матрицами 110
4.4. Выражения „Математики“ 113
4.5. Вычисление функций от списков и их элементов . . . 117
Упражнения 123

Часть II. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМПЬЮТЕРНОЙ АЛГЕБРЫ „МАТЕМАТИКА“ 125

Глава 5. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ 126
5.1. Функции, определяемые пользователем 126
5.2. Чистые и анонимные функции 129
5.3. Суперпозиция функций 132
5.4. Подмножества конечного множества 136
Упражнения 138

Глава 6. ПРОГРАММИРОВАНИЕ, ОСНОВАННОЕ НА ПРАВИЛАХ ПРЕОБРАЗОВАНИЙ 140
6.1. Глобальные и локальные правила преобразований . . 141
6.2. Шаблоны 148
6.3. Шаблоны в глобальных правилах преобразований . . 152
6.4. Шаблоны в локальных правилах преобразований . . 158
Упражнения 162

Глава 7. ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ 164
7.1. Составные выражения. Оператор **Do** 164
7.2. Условные операторы 167
7.3. Условные циклы 171
7.4. Функция **Module** 173
Упражнения 175

Глава 8. ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЙ 177
8.1. Значения, ассоциированные с символами 178
8.2. Атрибуты 179
8.3. Стандартный процесс вычислений 185
8.4. Выражения, вычисляемые нестандартно 187
8.5. Вычисление правил преобразований 191
Упражнения 194

Глава 9. РАЗРАБОТКА ПРОГРАММ 195
9.1. Контексты 195
9.2. Контексты и программы 201
9.3. Подгрузка программ 204
Упражнения 207
Глава 10. ВВОД И ВЫВОД ДАННЫХ 209
10.1. Ввод и запись данных в файлы 209
10.2. Обмен данными с другими программами 216
10.3. Форматирование выходных ячеек 218
Упражнения 224
Ответы и решения к упражнениям 226
Краткий справочник по встроенным функциям „Математики“ 232
Предметный указатель 252
Литература 258